The graPHIGS Programming Interface: Understanding Concepts

SC33-8191-04

The graPHIGS Programming Interface: Understanding Concepts

SC33-8191-04

Note

Before using this information and the product it supports, read the information in Appendix D, “Notices,” on page 325.

Fifth Edition (April 1994)

This edition applies to the graPHIGS API and to all subsequent releases of this product until otherwise indicated in new editions.

Contents

About This Book

This guide helps you understand the use of the graPHIGS API functions in your application to create, display, and interact with graphics data. It contains two parts: basic and advanced. Part 1 describes the *basics* of using the graPHIGS API and is especially suited for a first time user of the graPHIGS API. Included in every chapter of the Basic section are sample FORTRAN subroutines to give you hands-on experience with the graPHIGS API applications. For the experienced graPHIGS API programmer, Part 2 offers advanced functions and capabilities to further enhance your application program. It also expands upon some of the concepts introduced in Part 1. Used in conjunction with other graPHIGS API publications, this guide will help you create complete graphics application programs.

Who Should Use This Book

The graPHIGS Programming Interface: Understanding Concepts is useful to anyone who needs to understand the how the graPHIGS API functions in applications to create, display, and interact with graphics data. This book is useful to any graPHIGS API user—from new developers who need to learn basic graPHIGS concepts to experienced developers who want to learn about advanced functions and capabilities to enhance their existing application programs.

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
Italics	Identifies parameters whose actual names or values are to be supplied by the user.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Related Publications

Publications that relate to this product include:

- *The graPHIGS Programming Interface: Subroutine Reference*
- *The graPHIGS Programming Interface: Technical Reference*
- in *AIX 5L Version 5.2 AIXwindows Programming Guide*
- *AIX 5L Version 5.2 Installation Guide and Reference*
- *AIX 5L Version 5.2 Commands Reference*
- *AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs*

Part 1. Basic

Chapter 1. Introduction to the graPHIGS Programming Interface

What is the graPHIGS API?

The graPHIGS API(*) is a programming interface (subroutine package) used in the development of graphics applications. The graPHIGS API is based on the International Standards Organization (ISO) standard for Programmer's Hierarchical Interactive Graphics System (PHIGS). The graPHIGS API is a powerful graphics system that supports the definition, modification, and display of hierarchically organized graphics data.

It provides graphics application developers with a significant amount of additional function beyond the CORE, GKS, and PHIGS systems. The system adds new, more powerful concepts to provide a highly interactive, three-dimensional system that enhances the design and visualization process. The ability to organize graphic primitives into hierarchical structures makes it easy to edit, modify, and transform graphic entities.

The graPHIGS API provides programmers with the capability to design and code graphics applications that take advantage of high-function graphics devices. Using over 500 high-level graphic subroutine calls, programmers can develop applications in various programming languages.

The graPHIGS API also supports applications written to the ISO PHIGS standard interface. Applications may use calls (and the syntax) from both the graPHIGS API and ISO PHIGS interfaces, enabling access to the graPHIGS API functions, thus allowing expanded capabilities to ISO PHIGS applications.

The graPHIGS API system offers a powerful set of device-independent programming tools. The graPHIGS API decides whether to use local device capabilities or to have your central processing unit do the processing for your less intelligent workstations. The API makes the identical PHIGS function available in both mainframe and workstation environments.

Suitable application areas for graPHIGS include mechanical design, robotics, electronic design, textile design, process control, simulation, and a wide range of engineering and scientific uses.

Basic Concepts and Terminology

Graphical Resources

The graPHIGS API system consists of graphical resources with subroutines to control and utilize them. Graphical resources available to your application include:

- Structure Stores - collections of graphical elements (lines, text, etc) that help to create displayable objects
- Workstations - devices used to display the objects created by structure store elements. Workstations display the objects to a raster display or to data files, such as CGM or GDF format plot files. Workstations have input devices, such as keyboards and tablets, to provide input to your application.
- Font Directories - collections of displayable characters, typically used for different languages, appearances, or special-purpose user-defined symbols.
- Image boards - data collections for displaying images.

Subroutines

Types of subroutines available to your application include:

- Control subroutines - provide the basic control functions. These allow your application to open and close the graPHIGS API and allocate, share, control, and free graphical resources.

- Structure subroutines - provide control of structures, which are groupings of graphical elements. You can create, delete and modify structures. Modifications include changes to the whole structure content (such as emptying a structure), as well as changes to the elements in the structures (such as deleting or adding a single element in a structure).
- Element subroutines - provide the basic drawing facilities. These include primitives (such as lines, text, filled areas, curves, and surfaces) and their attributes (color, size, and linetype).
- Workstation subroutines - provide control of a workstation's facilities, such as setting a color table or view table entries.
- Display subroutines - provide the controls to display structure content on a workstation.
- Input subroutines - provide control of a workstation's input devices so that users may provide input to the application. For example, a user may want to pick a displayed object, provide text from a keyboard, or provide point or stroke (multiple points) input.
- Image subroutines - define image content and controls, such as color mappings and image display.
- Inquiry subroutines - provide your application information about the capabilities, state of resources and the system.

Resources and Capabilities

A typical use of the resources and capabilities of the graPHIGS API by your application might include the following:

- Create graphical data

  This involves creating structures containing elements and attributes that define displayed objects. Objects include:

  – Figures formed by lines and filled areas (such as a robot arm created by lines in different locations and colors)

  – Text such as labels, menus of options, and status information

  – Images.
- Open and control a workstation

  This involves identifying the current workstation and setting its values, (such as the color table), to those required for your application. This includes viewing information, which controls the parts of visible graphical data and their appearance on the display.
- Define the displayed content

  This entails associating structures of graphical data with a workstation, so that the workstation can draw the objects. The display content is modified by editing structures or changing the view tables used to display the graphical data.
- Accept user input

  This allows you to provide input to the application, typically to change the displayed objects. For example, to change an object a user might pick the object, select a choice provided by the function keys, or indicate a point or position using a tablet.

The graPHIGS API resources and facilities let you create a graphics application to display objects that a user can modify interactively. Your application can run in numerous environments, and inquiry subroutines provide information that enable your application to adapt to different hardware capabilities.

Common Terms

Following is a brief summary of common terms and their definitions used repeatedly throughout this manual.

Primitives:

The graPHIGS API defines a graphic system architecture that enables you to create, modify and display graphical objects. A sequence of elements define an object, including output primitives, attributes, and transformations. Basic output primitive elements include lines, markers, polygons, and text definitions.

Attributes:
Attributes define the characteristics of an output primitive. An attribute, for example, may define the color or size of a polymarker primitive.

Structures:
Graphical primitives and attributes group together to form structures. A structure may represent the geometry of an object, as well as information regarding the appearance of that object. Elements may be inserted into, or deleted from, structures at any time, in an operation called structure editing. This editing capability minimizes the need to redefine data in order to modify it. Structures may be related in a number of ways including geometrically, hierarchically, or characteristically, according to your application needs.

Input:
The graPHIGS API supports a wide range of input devices and provides the essential tools for application interaction. Input devices operating synchronously or asynchronously, relay information to the application, which in turn responds by defining, editing, or displaying the graphical data. The graPHIGS API supports six classes of input devices. These classes represent generic physical devices that differ from one another by the type of data they return to the application. Input device classes include the following:

1. Locator
2. Stroke
3. Valuator
4. Choice
5. Pick
6. String

The graPHIGS API supports three modes of interaction that allow you to request and obtain data from a logical input device. In **Request** mode, your application prompts for input and then waits until the operator either enters the requested input or performs a break action which terminates interaction. In **Sample** mode, your application obtains the current values of the input device by explicitly sampling it. In the **Event** mode, an asynchronous environment is established between your application and a chosen device. In this mode, both your application and any corresponding device operate independently of each other with the help of a centralized input queue.

Operating Modes:
The graPHIGS API supports three modes of interaction (Operating Modes) that allow you to request and obtain data from a logical input device.

1=REQUEST

Your application prompts for input and then waits until the operator either enters the requested input or performs a break action which terminates interaction.

2=SAMPLE

Your application obtains the current values of the input device by explicitly sampling it.

3=EVENT

An asynchronous environment is established between your application and a chosen device. In this mode, both your application and any corresponding device operate independently of each other with the help of a centralized input queue.

Workstations:
The term ″workstation″ refers to an abstraction of a physical graphics device. It provides the logical interface through which your application program controls physical devices.

The graPHIGS API provides an environment that supports multiple workstations. How your application interacts with a particular workstation depends on the interactive capabilities of that workstation and the design of your application.

The graPHIGS API supports three categories of workstations: INPUT, OUTPUT, and OUTIN. The capabilities of a workstation determine its category. For example, a INPUT workstation such as a digitizer provides only input, while an OUTPUT workstation, such as a plotter, generates only output. The OUTIN workstation, on the other hand, is an interactive design station that offers the capability of providing both input and output.

Inquiry Subroutines:

Inquiry subroutines allow the application programmer to access the program data contained in state lists, description tables, or structures. They are useful for determining both error conditions and device characteristics.

States:

The *system state* defines whether the graPHIGS API has been activated or deactivated, using the Open graPHIGS **(GPOPPH)** or Close graPHIGS **(GPCLPH)** subroutines respectively. No other subroutine calls can be accessed until the system is ″open″.

The *workstation state* defines whether a workstation has been activated or de-activated, using the Open Workstation **(GPOPPH)** or Close Workstation **(GPCLPH)** subroutines respectively. The graPHIGS API structure display subroutines can only be used if a workstation is ″open″.

The *structure state* defines whether the graPHIGS API display structure is ″open″ and able to be modified, or closed and unavailable for modification. A structure is opened and closed with the Open Structure **(GPOPST)** and Close Structure **(GPCLST)** subroutines. Graphics primitives and attributes can only be created if the structure state is ″open″.

Getting Started

The following is a sample graPHIGS API program to display an image of a house. It is coded using the FORTRAN language to run under VM or MVS using either a 5080 or 6090 workstation.

Accompanying the program is a sample of the output it will produce, as well as an explanation of its parts. In the chapters to follow you will be asked to modify this program and add more API functions to it, in addition to creating other assorted programs. At the end you will have a complete graPHIGS API sample program. Building the program step-by-step and then compiling, loading and running it, will help you understand the graPHIGS API.

Sample Program

```
* DECLARE VARIABLES
       INTEGER*4 STATUS,CHOICE
       INTEGER*4 WSID,STRID(5),VIEW1
       REAL*4 WINDOW(4),VIEWPT(4)
       REAL*4 HOUSE(12)
       DATA WSID /1/
       DATA STRID /1,2,3,4,5/
       DATA VIEW1 /1/
       DATA WINDOW /-100.0,100.0,-100.0,100.0/
       DATA VIEWPT /0.0,1.0,0.0,1.0/ *
       DATA HOUSE/0.0,0.0,0.0,40.0,30.0,70.0,
     *           60.0,40.0,60.0,0.0,0.0,0.0/
************************************************************************
* OPEN FUNCTIONS
       CALL GPOPPH('SYSPRINT',0)
       CALL GPCRWS(WSID,1,7,'IBM5080','5080     ',0)
************************************************************************
* VIEW DEFINITION CALL
       GPXVR(WSID,VIEW1,14,VIEWPT)
       CALL GPXVR(WSID,VIEW1,16,WINDOW)
       CALL GPXVCH(WSID,VIEW1,1,8,2)
************************************************************************
* DATA CREATION
       CALL GPOPST(STRID(1))
       CALL GPPL2(6,2,HOUSE)
       CALL GPCLST
************************************************************************
* DATA DISPLAY
       CALL GPASSW(WSID,1)
       CALL GPARV(WSID,VIEW1,STRID(1),1.0)
       CALL GPUPWS(WSID,2)
************************************************************************
* INPUT FUNCTIONS
       CALL GPRQCH(WSID,1,STATUS,CHOICE)
************************************************************************
* CLOSE FUNCTIONS
       CALL GPCLWS(WSID)
       CALL GPCLPH
       STOP
       END
```

The following figure shows the drawing displayed when the sample program is executed:

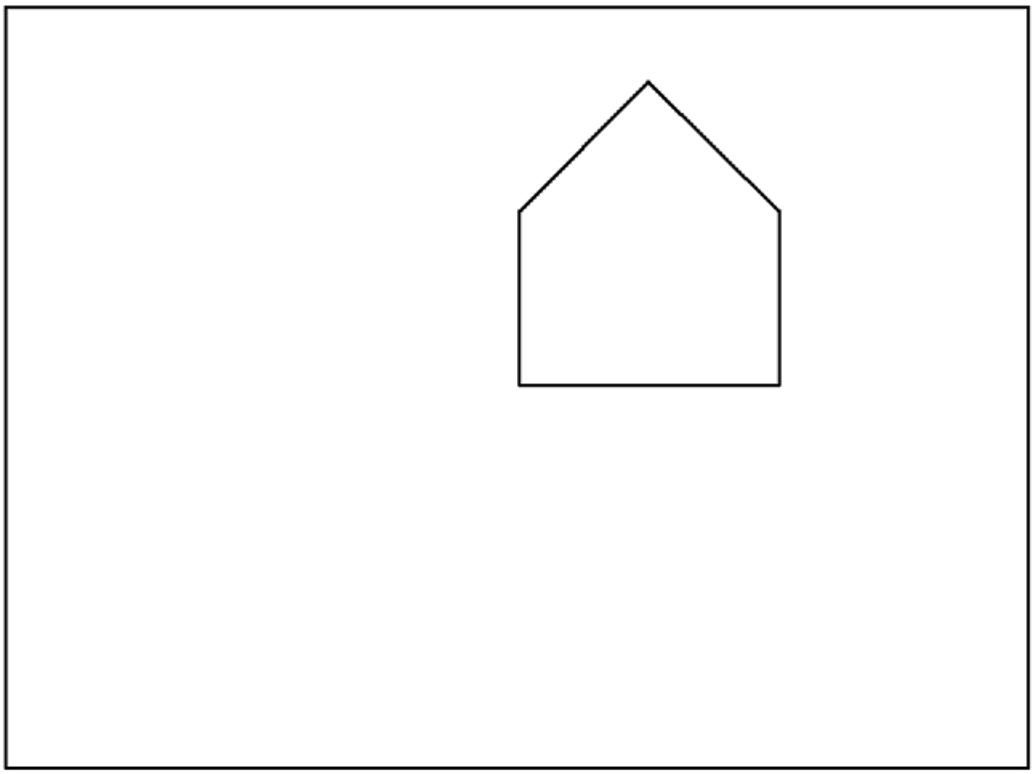


Figure 1. House Created by Sample Program. This illustration uses a solid line to represent the basic shape of a house with a pitched roof.

To simplify the explanation, we have divided it into several parts. Following is a description of the various parts and their functions:

1. `DECLARE VARIABLES` is closely related to the language you are using to code your sample program. Depending on the language, you will probably need to declare the type and initialize the different variables of your application program before using them. In this section some of the parameters of the graPHIGS API subroutines are defined, as well a the coordinates of the drawing displayed by the program.
2. `OPEN FUNCTIONS` enable you to access all the available graPHIGS API application subroutine calls as well as define the workstation that will display the image. The sample program defines a 5080 type workstation and assigns it an identifier (*wsid*) of 1, which will be used when referring back to the workstation. If you are using a 6090 type workstation, your application will automatically access it as a 5080 type workstation.

   If you are using a workstation other than a 5080 or 6090, then replace the **GPCRWS** statement with one of the following:

   - If you are using a GDDM type workstation, then use:

     ```
     CALL GPCRWS(WSID,1,1,'*','GDDM    ',0)
     ```

   - If you are using an AIX type workstation, use:

     ```
     CALL GPCRWS(WSID,1,1,'*','X       ',0)
     ```

3. `VIEW DEFINITION CALL` is the part of the program where your application defines where on the display screen the drawing will be shown (*viewpt*), and what portion of the image will be displayed (*window*). The sample program uses all of the screen for display. Also, the drawing has been defined in a coordinate system from -100.0 to 100.0 in both the X and Y directions. These coordinates define a view window in the Workstation State List (WSL). Your application also needs to activate this view, **(GPXVCH)**, in order for its content to be displayed.

4. `DATA CREATION` is the section of your program that creates the drawing. Drawings, or graphical models, are created using *structures*, the heart of the graPHIGS API data organization. Each structure is assigned an identifier. The sample program assigns the structure an identifier (*strid*(1)) of 1, which is used whenever referring to the structure.
5. `DATA DISPLAY` routes the drawing to the appropriate workstation for display. Before displaying your drawing, a structure store must first be associated to all views on a workstation **(GPASSW)** After this association is done, the structure from the structure store that has your drawing (*strid*(1)) is then associated to view 1 (*view1*) on the workstation (*wsid*). Finally, to display your drawing, you must update the workstation **(GPUPWS)**
6. `INPUT FUNCTIONS` allow you to interact with your application program. The example program simply displays the house and waits for a function key to be pressed before proceeding.
7. `CLOSE FUNCTIONS` enable your application to free the workstation and other resources used while running your application, or close the graPHIGS API system upon completing your application. The last two instructions in the sample program close the workstation and the system.

For additional information and detailed coding of specific subroutine calls, refer to *The graPHIGS Programming Interface: Subroutine Reference*, SC33-8194.

Chapter 2. Accessing the System

The graPHIGS API is a powerful set of device-independent programming tools that allows you to design and code graphics applications that take advantage of high-function graphics devices. This chapter reviews how to access these programming tools.

Open Subroutines

The graPHIGS API is divided into two distinct parts: the graPHIGS API shell and the graPHIGS API nucleus. The graPHIGS API *shell* is tightly coupled to your application and does the syntax checking and building of the graphics datastream. The graPHIGS API *nucleus*, on the other hand, may be tightly coupled to your application also, or may run separately. The nucleus manages resources such as the structure store and the workstations. You will learn more about the relationship of the graPHIGS API shell and nucleus in the second part of this manual.

As shown in the previous sample program, to access the graPHIGS API subroutines, your application must ″open″ the system.

Opening the System

After issuing the Open graPHIGS **(GPOPPH)** subroutine call, all the graPHIGS API subroutines are available to your application. This subroutine also makes various stored information, maintained in internal description tables and state lists, available to your application. Due to the centralized storage for graphics data, your application can create and modify graphics data at this point without needing to open a workstation.

At the time your application issues the Open graPHIGS subroutine, a graPHIGS API shell is created. By default, the graPHIGS API will also perform the following functions:

- connect to a nucleus with nucleus identifier of 1
- create a structure store with identifier of 1 on nucleus 1
- select structure store 1 as the current structure store

Default creation of a nucleus and structure store is provided so that applications written for Version 1 of the graPHIGS API will continue to run without modification. The default processing of the nucleus and structure store can be changed through an External Defaults File (EDF), or through the second parameter of the **GPOPPH** subroutine.

In the sample program, the statement:

```
CALL GPOPPH('SYSPRINT',0)
```

opened the system and defined the environment explained above. Through this subroutine call you can also define the name of a file that will be used by the graPHIGS API as an error log. In the sample program we are naming this file 'SYSPRINT' This file will help with debugging the application.

For more information concerning the use of external defaults files and the use of the second parameter of the **GPOPPH** subroutine, refer to the *The graPHIGS Programming Interface: Technical Reference*

Opening the Workstation

To display a picture that is generated from your application's graphics data, a workstation must be open. Workstations are resources owned by the nucleus. When your application issues the Create Workstation **(GPCRWS)** subroutine, it opens a workstation in a graPHIGS API nucleus and attaches it to a graPHIGS API shell. The following activities also take place:

- a Workstation State List (WSL) and actual Workstation Description Table (WDT) are created and initialized for that workstation
- a physical connection is established with a physical device
- the display surface is cleared

In the sample program, the statement:

```
CALL GPCRWS(WSID,1,7,'IBM5080','5080    ',0)
```

opens a 5080 type workstation and establishes a connection between the workstation and the sample program. The workstation is given the workstation identifier `WSID`, which is used later in the program to identify the workstation.

When a workstation is opened the graPHIGS API creates and initializes an Actual Workstation Description Table (WDT) based on the workstation type parameter of the Create Workstation **(GPCRWS)** subroutine and the actual capabilities of the workstation. The graPHIGS API also creates a generic WDT which describes the capabilities of a generic workstation of the specified type.

An example of the values inserted into the actual WDT that may differ from the generic WDT include:

- the workstation's color capabilities
- the available input devices
- the display size

To make your application device independent, you must access the actual WDT to determine the characteristics and capabilities of the workstation your application is using. To do this, first open the workstation, then determine the name of the actual WDT by using the Inquire Realized Connection and Type **(GPQRCT)** subroutine. Then use the workstation type parameter returned by **GPQRCT** to inquire the actual workstation capabilities. Inquiry subroutines are discussed in more detail in subsequent chapters.

Close Subroutines

The close subroutines enable your application to end its use of a particular workstation, the system, or both.

In the sample program the following two statements closed the workstation and the system:

```
CALL GPCLWS(WSID)
CALL GPCLPH
```

Closing the Workstation

When your application issues the Close Workstation **(GPCLWS)** subroutine, the graPHIGS API disconnects the target workstation and releases memory and other resources allocated for the workstation. **GPCLWS** also clears the input queue of any events from the given workstation.

Closing the System

When your application issues the Close graPHIGS **(GPCLPH)** subroutine, each resource, such as an open workstation, is closed. **GPCLPH** also invokes a set of routines that close the system. This frees all of the state lists, description tables, and associated system resources.

Chapter 3. Structures

The graPHIGS API stores its graphical data into structures. In some cases, where the complexity of the drawing or application demands it, the structures can be linked together in a hierarchical type of network.

Although structures can be useful for organizing the graphical data, they are not required in order to display a picture. Graphical data can be sent directly to a workstation for display without storing the data in structures. See Chapter 14. Explicit Traversal Control for a description of this processing.

Creating Structures

Your application uses structures to establish relationships between the basic elements of your graphics data. This relationship is based on a hierarchical network, which enables structures to reference or execute other structures. Once a structure has been created, it can easily be modified by inserting or deleting structure elements.

All structures are defined in a structure store. At the time your application issued the Open graPHIGS subroutine, a structure store was created and selected as the current structure store. All structures created after this subroutine call will be defined in this structure store.

In the Sample Program, you created the following structure to draw the house:

```
CALL GPOPST(STRID(1))
CALL GPPL2(6,2, HOUSE)
CALL GPCLST
```

Each data item within the structure is called a *structure element*. The sample program structure has only one element, called an output primitive, which allows you to draw the house. Structure elements vary greatly in their content and organization, enabling your application to define, organize, and control all parts of a graphical model. The following is a list of element types that may be contained in a structure:

- Output primitives
- Attribute settings
- Modeling transformations
- Invocation of a sub-structure
- Labels that delineate specific elements
- Application specific data
- Pick information
- Class name elements

Sample Program

Now, refer back to the sample program and modify it according to the following statements:

1. Add the following statements to the DECLARE VARIABLES section:

```
REALx4 MATRIX(9)
INTEGERx4 POST
DATA MATRIX /1.0,  0.0, 0.0,
             0.0,  1.0, 0.0,
           -30.0,-35.0, 1.0/
DATA POST /2/
```

2. Replace the statements of the DATA CREATION section with the following:

```
CALL GPOPST(STRID(1))
CALL GPMLX2(MATRIX, POST)
CALL GPLT(2)
CALL GPPL2(6,2,HOUSE)
CALL GPCLST
```

Recompile, load and run the program after you have input these modifications. The following figure shows the picture you will get when the modified sample program is executed:

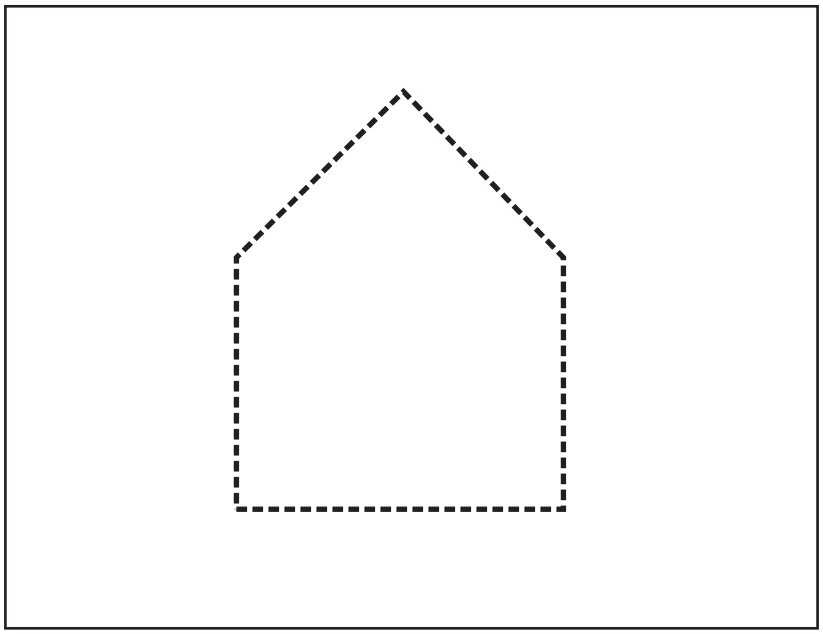


Figure 2. Modified Sample Program Picture. This illustration uses a dashed line to represent the basic shape of a house with a pitched roof.

As you can see, some new elements have been added to the structure. The line type and position of the house have been modified. One structure element modified the line type attribute setting of the output primitive and another applied a modeling transformation to it. In the chapters that follow, you will learn in more detail how to define each structure element. In the previous structure you were using default values. Information on default settings can be found in *The graPHIGS Programming Interface: Technical Reference*.

Opening a Structure

If your application needs to create a structure or access the content of an existing structure, it issues an Open Structure **(GPOPST)** subroutine. Your application can assign each structure a unique structure identifier to aid in the differentiation between structures. To open an existing structure, your application specifies the existing structure's identifier in the **GPOPST** subroutine.

When your application tries to open a non-existent structure by specifying an unused structure identifier, the API creates a new empty structure. Other ways in which the system implicitly creates a structure are:

- by inserting a reference to a non-existent structure into an open structure
- by associating a non-existent structure with a workstation or by adding the non-existent structure to a view (discussed in Chapter 6. Displaying Structures)

When a structure is open, it has a conceptual *current element pointer*. The element pointer is a mechanism that lets your application locate a position within the open structure.

When an application program opens a structure, the element pointer points to the end of the structure. In an empty structure, the element pointer points to the conceptual element zero, which is null.

As elements are added, the system increments the element pointer to the most recently added element. The system also assigns a sequence number to each element. The first element is always 1, the second is always 2, and so on. The following figure shows how your application adds elements to a structure and how the system increments the current element pointer:

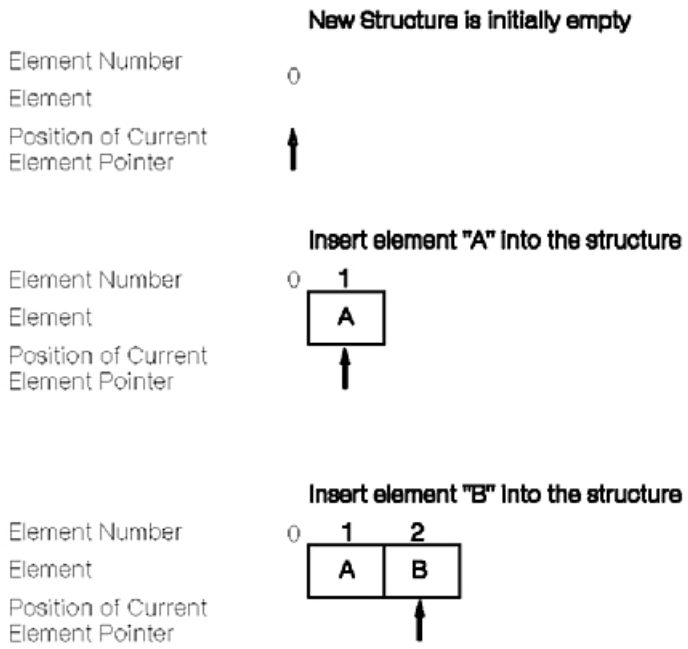


Figure 3. Inserting Elements Into an Empty Structure. This diagram shows the element pointer, represented by an arrow, in three different scenarios: empty structure, structure with element A inserted, and structure with element B inserted. In the first scenario, the arrow points to nothing. In the second scenario, the arrow points to the letter A, which is the only element in the structure (element 1). In the third scenario, the arrow points to the letter B (element 2), which has been added after the letter A in the structure.

Structure Hierarchies

As previously discussed, the graPHIGS API gives your application the ability to group graphic elements within structures. It also enables your application to organize structures into hierarchical structure networks.

In many instances, a hierarchy can effectively represent the relationships among the parts of a model. This capability provides a way for your application to functionally attach (assemble) individually defined primitives. For example, your application can functionally attach a table's legs to its top. The following figure shows the assembly of a graphics model from individually defined primitives:

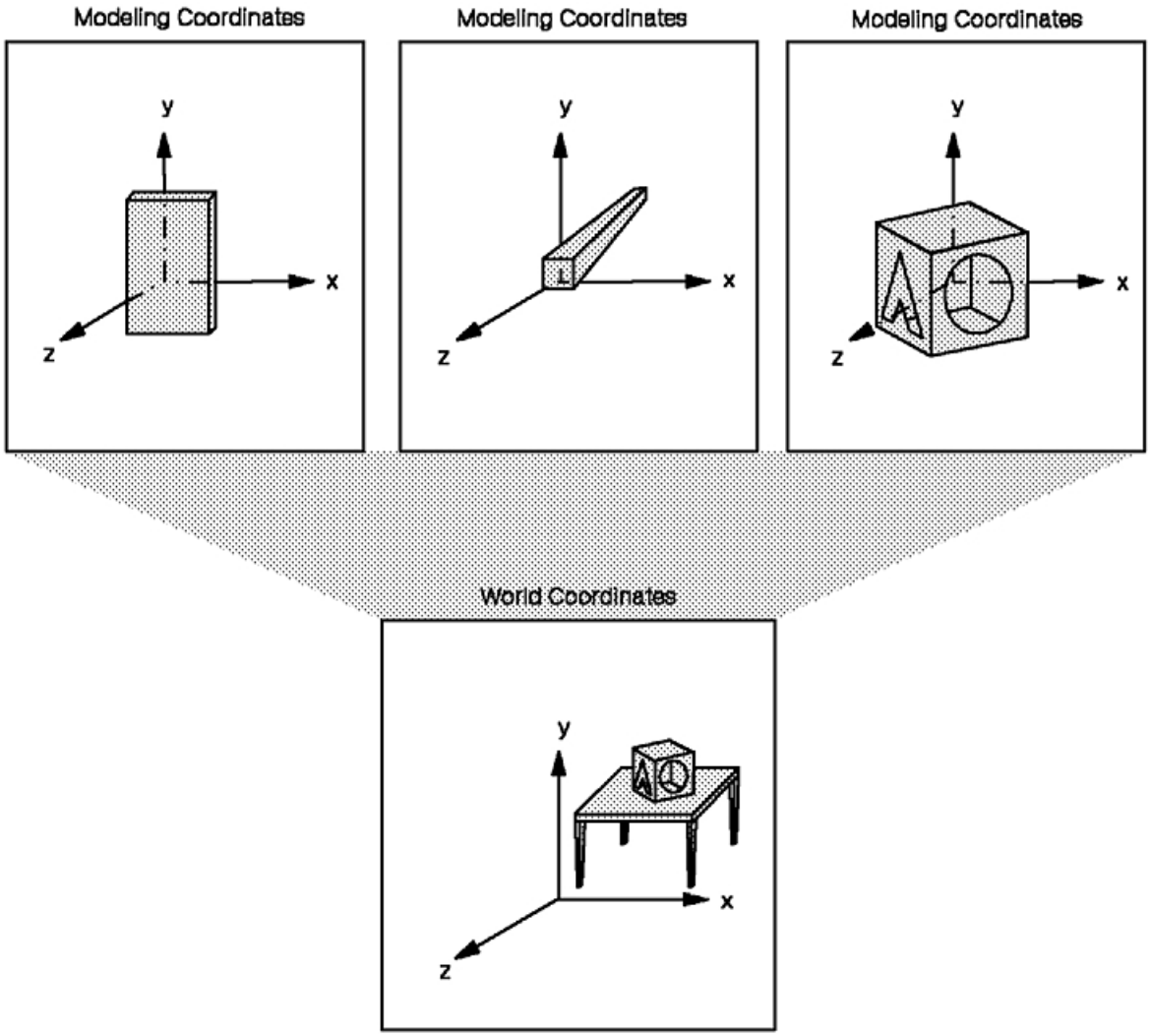


Figure 4. Using Modeling Transformations to Assemble Graphics Models. This illustration depicts the assembly of a scene from its parts: a table top, a table leg, and a box in Modeling Coordinates. The scene consists of one top and four legs transformed into a table in World Coordinates, with the box resting on top.

In order to establish hierarchies, the graPHIGS API provides a structure element that invokes other structures. To create a structure element that invokes another structure, use the Execute Structure **(GPEXST)** subroutine.

One way to organize the table's graphics data is to imbed a structure reference in the tabletop structure that invokes the leg's structure. The following figure depicts one of many ways to organize the table's top and legs hierarchically:

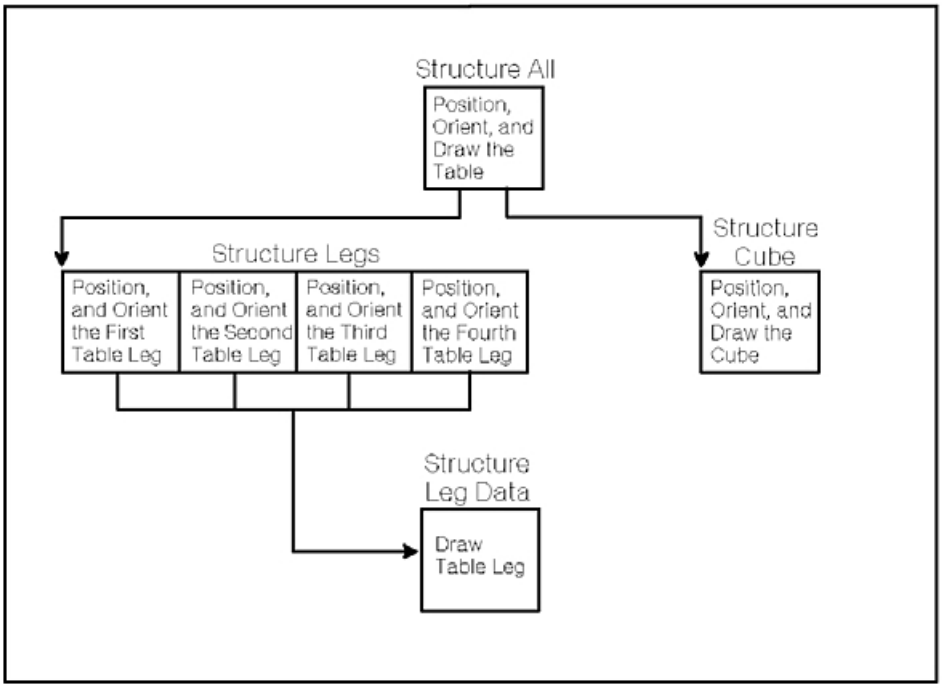


Figure 5. Hierarchical Structure Relationships. This diagram depicts a hierarchical organization of the table model described in figure above. The root structure positions, orients, and draws the table top, with links to the Legs and Cube structures. The Cube structure positions, orients, and draws the cube. The Legs structure positions and orients each leg, with links to the (individual) Leg structure. The Leg structure draws a single table leg.

Note: The depicted organization is only intended to illustrate hierarchies. Other structure organizations may perform better in such cases.

Hierarchically organized structures also provide efficient use of storage. Your application needs to define common parts only once, then reference them as many times as necessary. Rather than defining duplicate leg structures, your application invokes the same leg structure many times.

The boxes shown in the figure, *Hierarchical Structure Relationships*, represent structures. Their organization enables your application to define the position of each leg *relative to* the tabletop's position. If your application repositions the top, the legs **inherit** that new position. The attached legs are automatically repositioned relative to the new position of the top.

At this point, you are ready to modify the sample program and create another structure. This new structure will add a door to the house.

Sample Program

Modify your house sample program according to the following statements to create a new structure:

1. Add the following to the DECLARE VARIABLES section:

```
REALx4 DOOR(10),TRANSD(9)
DATA DOOR /0.0,0.0,0.0,20.0,10.0,20.0,
x          10.0,0.0,0.0,0.0,0.0/
DATA TRANSD / 1.0,0.0,0.0,
x              0.0,1.0,0.0,
x             10.0,0.0,1.0/
```

2. Add a new stucture to the DATA CREATION section using structure identifier 2. Code these statements after the **GPCLST** of the polygon structure.

```
CALL GPOPST(STRID(2))
CALL GPMLX2(TRANSD,POST)
CALL GPPL2(5,2,DOOR)
CALL GPCLST
```

   At this point you need to relate the door structure with the house structure. In order to indicate this, add an invoke structure statement in structure 1.

3. Add the following statement after the CALL GPPL2 statement of structure 1:

```
CALL GPEXST(STRID(2))
```

Now run the sample program. Note that although no line type was specified for the door, it is also a dashed line. Being a child structure of structure 1, the door has inherited its linetype attribute. To assemble the door with the house, your application program makes use of modeling transformation (TRANSD) which moves the door to the correct position within the house drawing. The house modeling transformation (matrix) is also inherited by the door and allows it to move with the house to the center of the screen. See the following figure:

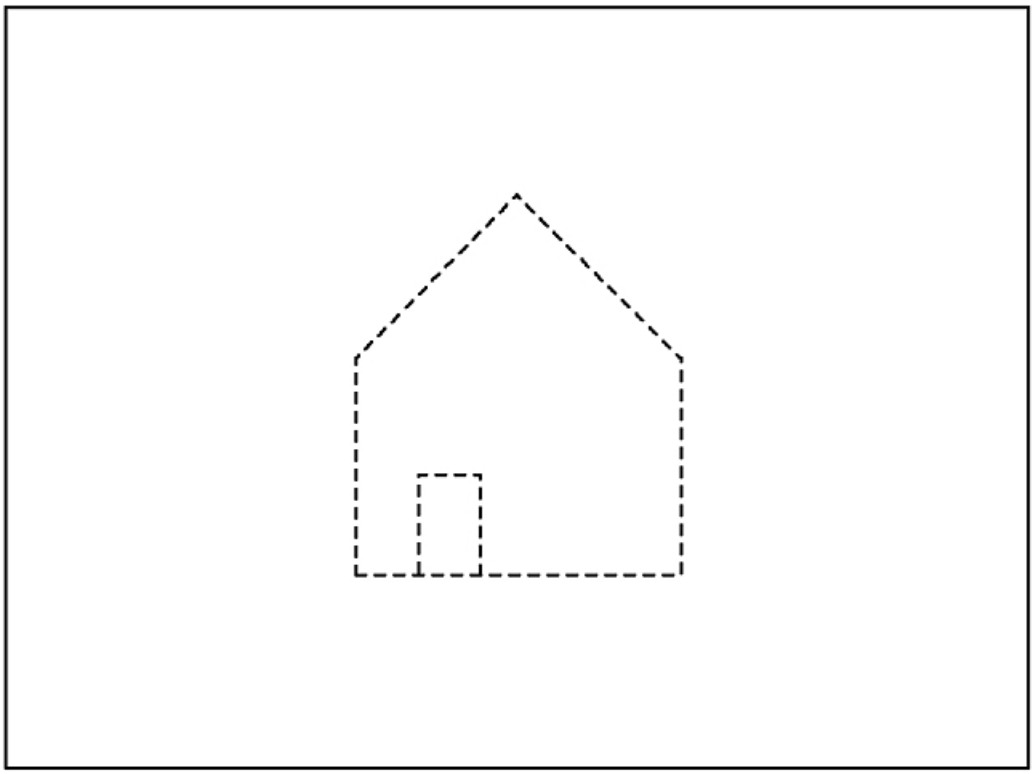


Figure 6. Modified Sample Program Output. This illustration uses a dashed line to show the basic shape of a house with a pitched roof and a door.

Hierarchies impose the passing of attributes from parent to child structure. If there was another child structure which drew a polyline, it would inherit the style and color attributes of its parent structure, in this case structure 1.

The ability to inherit attributes and transformations reduces the need for editing every related structure in a complex model. It also enables your application to easily alter the relationships among models and between parts of models.

Since there are many ways to establish hierarchical structure networks, the actual organization depends on the needs of your application.

Closing Structures

Operations on the currently opened structure end when your application closes the structure with the Close Structure **(GPCLST)** subroutine.

The graPHIGS API provides operations that affect entire structures, and permit your application to inquire and modify the content of structures. Those topics are discussed in Chapter 8. Structure Editing.

Chapter 4. Structure Elements

This chapter reviews the basic data elements that form structures. It presents output primitive elements such as lines, markers, polygons, and text. Along with a discussion of how to define each primitive, this chapter describes how to control their appearance through attribute settings and how to use modeling transformations to affect the size, shape, location, and orientation of primitives.

Output Primitive Attributes

The appearance of graphical data is controlled through output primitive attributes. Color, interior style, line type, are some of these attributes. For example, in Chapter 3, the line type of the house in the sample program was defined using an attribute-setting subroutine.

This section provides a general discussion of output primitive attributes. Later on, there will be discussions of specific attributes associated with each primitive.

When a primitive subroutine or an attribute-setting subroutine is called, a structure element is created. The effect of the element on the graphical model is not seen immediately. The data to perform the requested operation is placed in the graPHIGS API storage area, but the effect of the element is seen only after the structure containing the element is processed and displayed. This processing of a structure at display time (called *structure traversal*) is discussed further in Chapter 6. Displaying Structures.

The graPHIGS API provides two ways for your application to assign an attribute to an output primitive:

- directly from an individual setting
- by way of an index into an attribute table called a bundle table

For your application to control where a primitive obtains its attributes, the graPHIGS API provides Attribute Source Flags (ASFs). ASFs can be thought of as switches within a structure that indicate whether the primitive obtains an attribute directly from a *conceptual register* or from an entry in a workstation bundle table.

Each workstation maintains internally a set of *conceptual registers* that contain the current values of each attribute at display time. Your application can set the ASF of each attribute at any location within a structure to

```
1=BUNDLED
2=INDIVIDUAL
```

through the Attribute Source Flag Setting **(GPASF)** subroutine. Initially, all ASFs are set to `INDIVIDUAL` in order to obtain attributes directly from the conceptual attribute registers.

The system provides an ASF for each non-geometric primitive attribute. Some attributes, like geometric character height, are expressed in Modeling Coordinates (MC) and are affected when transformations are applied to the Modeling Coordinate System (MCS). Others, like color, have no effect on the output primitive orientation. The first type of attribute is a geometric attribute and cannot be included in a bundle table. The second is a non-geometric primitive attribute and can be bundled.

Special types of attributes that control detectability, highlighting, and visibility of all primitives are introduced in Highlighting, Detectability, and Invisibility Class Specification.

Individual Attributes

The graPHIGS API provides attribute setting structure elements that enable your application to specify the value of each attribute individually. This method of attribute specification is workstation-independent. By using the proper structure elements, your application might change, for example, text color from red to green by specifying a color index.

In the Sample Program, the line type used to draw the house is `1=DASHED`**.** This attribute is set by using the Set Linetype **(GPLT)** subroutine with a line type index of 2. Try changing this index to different values.

Internally, each workstation maintains a set of *conceptual registers* that contain the current values of each attribute during display traversal. The graPHIGS API lets your application change the conceptual attribute settings through attribute setting structure elements. When an output primitive is encountered during display traversal, the API uses the current values in the registers to draw that primitive.

When beginning the traversal of a structure, the conceptual attribute registers contain default values. If the application has defined no attributes, as is the case of the Sample Program, these default values are then used to display the graphical data. Also, in hierarchical structure networks, if the child structure has no attribute settings, the current values of the attributes (either default or defined by its parent structure) are used.

Bundled Attributes

The graPHIGS API gives your application the ability to group attribute settings into tables of output primitive attributes called *bundle tables*. Each workstation has its own bundle tables. This lets your application use a common index across workstations to specify attribute settings that are defined in workstation-dependent tables.

For each type of bundle table, there is a Set Representation subroutine call to set or change the bundle table contents. Please note that this subroutine call does not create a structure element.

The information in bundle tables exists independently of your application's graphics data. In order to establish a connection between an output primitive and an entry in the bundle table, your application specifies an index that points to that entry through a set bundle table index structure element.

Each output primitive uses one or more bundle tables. For example, polygons use the edge and interior bundle tables, while polylines use only the polyline bundle table.

Refer back to the modified Sample Program and modify it again as follows:

Sample Program

Refer to your house sample program and modify it as follows:

1. Add the following statements to the DECLARE VARIABLES section:

```
INTEGERx4 ATLIST(3),ATFLAG(3),COLOR(2)
DATA ATLIST /1,2,3/
DATA ATFLAG /1,1,1/
DATA COLOR /1,2/
```

2. In the DATA CREATION section, add the following before the GPOPST(STRID(1)) statement:

```
CALL GPXPLR(WSID,1,1,1)
CALL GPXPLR(WSID,1,2,2.0)
CALL GPXPLR(WSID,1,3,COLOR)
```

3. In the same section, replace the lines which created structure 1 with the following statements:

```
CALL GPOPST(STRID(1))
CALL GPMLX2(MATRIX, POST)
CALL GPASF(3,ATLIST,ATFLAG)
CALL GPPLI(1)
CALL GPPL2(6,2,HOUSE)
CALL GPEXST(STRID(2))
CALL GPCLST
```

Recompile, load and run the program after making these modifications. The image displayed should be red with a solid line type.

Instead of defining the attributes individually, you are now using bundle table definitions. In the first statement of the DATA CREATION section, you have stored attribute settings in the first table index entry of the polyline bundle table of the workstation **(GPXPLR)** Then, in the structure, using the Attribute Source Flag Setting **(GPASF)** subroutine, you have set the line typ e, width and color of the polyline to BUNDLED. Use the Set Polyline Index **(GPPLI)** subroutine to set the attribute for the polyline.

By specifying attribute settings through table entries, your application can associate all of the values in a particular entry with an output primitive by simply specifying the index to that entry. This saves your application from redefining every attribute of commonly used output primitives. If your application sets the index such that it points beyond the last table entry, the system, during traversal, uses table entry 1.

Using an index to obtain attribute values has an additional value. You can set the bundle tables of different workstations such that the same index entry compensates for the workstation's differences. For example, line types can be used to differentiate between lines on a workstation with a monochrome display, and color can be used to do the same on a workstation with color display.

Bundling also provides a simple method of standardizing multiple instances of a model's appearance. For example, an application program modeling a bolt may put the structure in many locations. In each instance, the application program provides an index to a predefined entry in the polyline bundle table just prior to invoking the bolt structure.

The following table depicts a sample polyline bundle table:

Table 1. Sample Polyline Bundle Table

Index	Linetype	Linewidth	Color Index
1	1	1.00	2
2	2	0.50	8
3	2	2.00	5

Table 1. Sample Polyline Bundle Table (continued)

Index	Linetype	Linewidth	Color Index
4	1	1.50	3
5	3	1.00	4
6	1	1.00	9
7	1	1.00	10

Mixing Individual and Bundled Attribute Specifications

For each attribute, the ASF specifies whether a primitive obtains an attribute directly from an individual setting or through an entry in a workstation bundle table. The graPHIGS API permits your application to change many ASFs at once by specifying multiple attributes and their corresponding ASF values in the same **GPASF** subroutine.

Sample Program

Refer back to the Sample Program as you have it after adding the previous modifications. Do the following:

1. Change the following statement in the `DECLARE VARIABLES` section:

```
DATA ATFLAG /2,1,1/
```

2. Modify the structure 1 statements of the `DATA CREATION` section so they look like the following:

```
CALL GPOPST(STRID(1))
CALL GPMLX2(MATRIX, POST)
CALL GPASF(3,ATLIST,ATFLAG)
CALL GPPLI(1)
CALL GPLT(3)
CALL GPPL2(6,2, HOUSE)
CALL GPEXST(STRID(2))
CALL GPCLST
```

Recompile, load, and run the program after you have input these modifications. You will notice the house and door are no longer dashed; they now appear dotted.

You are using mixed attribute specifications. Your application has the ASF individual flag on for the line type (attribute identifier 1). Although the polyline bundle entry you have created specifies line type of `2=DASHED`, due to the ASF flag setting for line type, the application obtains its line type attribute setting from the `GPLT(3)` subroutine call instead, which is set to `3=DOTTED`.

In the case of the sample program we are using one bundle table; the polyline bundle table. In the case of polygons, two bundle tables are used; the interior bundle table and the edge bundle table. For example, your application might use the **GPASF** subroutine to set the polygon output primitive's:

- Edge line type to bundled
- Interior style to bundled
- Interior color index to individual

After using **GPASF** to set the edge line type and interior style to `1=BUNDLED`, your application can specify which entry to use in the edge bundle table and interior bundle table using the Set Edge Index **(GPEI)** and Set Interior Index **(GPII)** subroutines respectively.

In the above example, the system obtains the edge line type from the edge bundle table and the interior style from the interior bundle table. The system ignores the color value entry from the interior bundle table and obtains it directly from its conceptual register, as specified by the interior color ASF.

Please note that if a parent structure exists, indexes to bundle table entries and individual (2=INDIVIDUAL) settings are both inherited.

How Workstation Capabilities Affect Realized Attribute Values

Because the graPHIGS API provides the tools necessary to develop workstation-independent programs, you may not need to change the attribute specifications in your application program to accommodate the differences between workstations. The graPHIGS API automatically maps workstation-independent attribute specifications to the closest workstation-dependent attribute specifications that can be realized at the target workstation. This section addresses the way in which workstation capabilities affect realized attribute values for scale factors and indexed tables.

Scale Factor Specification

At draw time, the system maps all application-specified, scale-factor attribute values to the closest scale factor value supported by the workstation. Then it multiplies that scale factor by the workstation's nominal value to obtain the primitive's size in Device Coordinates (DC). Your application can inquire the system defaults for these values. It can also inquire the workstation's supported values while the workstation is open.

The following table shows a case in which the workstation supports four scale factor values (0.5, 1.0, 2.0, and 3.0) with a nominal size of 2 Device Coordinates (DCs):

Table 2. Example of Workstation Support of Four Scale Factor Values

Application-Specified Scale Factor	Closest Scale Factor That is Available on This Device (DSF)	Output Device Nominal Size of Primitive (N)	Size of Primitive (DSF x N)
1.0	1.0	2	2
0.2	0.5	2	1
0.7	0.5	2	1
1.5	2.0	2	4
7.0	3.0	2	6

Indexed-Table Specification

Each workstation has its own bundle and color tables. In order to remain workstation-independent, your application uses indexes to specify entries into these tables. The workstation simply applies the specified index value to its own table. For example, the same workstation-independent index value that indicates blue on a color display can indicate a corresponding greyshade on a monochrome display.

The system provides default values for all workstation table entries when it opens the workstation. These defaults can be found in *The graPHIGS Programming Interface: Technical Reference*, or by using the appropriate inquire subroutines.

Color Specification Tables

Colors in the graPHIGS API are assigned using tables of color specifications. A color table has entries that specify the values of the red, green and blue intensities defining a particular color. Each color table is workstation-dependent. A color is assigned to a primitive or a view by specifying an index into a workstation's color table.

The Set Color Representation **(GPCR)** subroutine can be used to load entries into the default workstation color table. The entries are loaded using the **(GPCR)** subroutine and are interpreted in the current workstation color model (discussed below). All workstations supported by the graPHIGS API have a default color model of RGB. Color table entries can be inquired using the Inquire Color Representation **(GPQCR)** subroutine. Notice that values returned by **GPQCR** are returned in the current workstation color model.

The graPHIGS API supports four methods for specifying color values. The four supported color models are:

1=RGB
(Red, Green, Blue)

2=HSV
(Hue, Saturation, Value)

3=CMY
(Cyan, Magenta, Yellow)

4=CIELUV
(Commission Internationale de l'Eclairage system based on luminance and chromaticity coordinates)

The color model is a specification of a three-dimensional color coordinate system within which each displayable color is represented by a point. A workstation's color capabilities including whether color is supported, the default color model and the color palette size can be inquired using the Inquire Actual Color Facilities **(GPQACF)** subroutine. The current color model can be inquired using the Inquire Color Model **(GPQCML)** subroutine.

Each workstation has a preferred color model. For example, the IBM 5080 graphics system uses an RGB model. The initial table definition and subsequent entries are stored in this preferred model. If the application changes the current color model, all subsequent entries are converted from the new color model to the preferred model. Changes to the color model are not retroactive. In other words, all entries loaded before the color model change are not affected.

For each application defined color value, a workstation uses the available color which most closely matches the application defined values.

Sample Program

Look back at the previous version of the Sample Program and do the following.

1. Add the following statements to the `DECLARE VARIABLES` section:

```
REALx4 CTABLE(3)
DATA CTABLE /0.40,0.25,0.25/
```

2. Change the following statement in the `DECLARE VARIABLES` section:

```
DATA ATFLAG /1,1,2/
```

3. Add the following statement to the `DATA CREATION` section, before the `CALL GPOPST(STRID(1))` subroutine call:

```
CALL GPCR(WSID,6,1,CTABLE)
```

4. Modify the statements of the `DATA CREATION` section for structure number 1, so they look like the following:

```
CALL GPOPST(STRID (1))
CALL GPMLX2(MATRIX,POST)
CALL GPASF(3,ATLIST,ATFLAG)
CALL GPPLI(1)
CALL GPPLCI(5)
CALL GPPL2(6,2,HOUSE)
CALL GPEXT(STRID(2))
CALL GPCLST
```

5. In the same section, after the `CALL GPMLX2(TRANSD,POST)` of structure number 2, add the following statement:

```
 CALL GPPLCI(6)
```

Recompile, load and run the program after you have input these modifications. The color of the house is yellow, but now the door is brown.

By issuing the Set Color Representation **(GPCR)** subroutine, your application has redefined the color table of your workstation. Index six of your table now contains the color brown. Structure 2, the child structure, uses index six to set the color attribute of the door polyline. By setting its own color attribute, the child structure does not inherit its parent structure color setting. Try creating your own color table by specifying different values of red, green and blue intensities.

Basic Output Primitives

The graPHIGS API provides five basic output primitives that your application can use to define the parts of a graphics model. The graPHIGS API discusses advanced output primitives in Part Two of this book. Basic primitives include:

- Polylines
- Polymarkers
- Polygons
- Geometric Text
- Annotation Text

These output primitives and their attributes are discussed in the following sections.

Polyline

Polylines provide a means for drawing lines of various types, widths, and colors. Drawing a polyline is like using a pen to draw a line on a piece of paper without lifting the pen from the page. If you need or want to break the line, you simply create a new polyline, or use the advanced disjoint polyline output primitive discussed in Disjoint Polyline Primitive.

Polyline Output Primitive

The polyline output primitive lets your application construct connected line segments in two or three dimensions.

For each polyline, your application provides a series of points in order to define its line segments. The graPHIGS API accepts, as input, *one* array into which your application has specified the X, Y, and Z modeling coordinate values for each point. The array is of a *list of points* that might look like: 1x,1y,1z,2x,2y,2z,...Nx,Ny,Nz.

The Polyline **(GPPL2** and **GPPL3)** subroutines are used to create two- and three-dimensional polyline structure elements, respectively. At draw time, the system interprets the first X, Y, (and Z with the **GPPL3** subroutine) coordinates as the starting point for the first line segment. The system then draws a line from the first coordinate point to the next coordinate point, then to the next, and the next, and so on.

With each of these subroutines, your application specifies:

- the total number of points in the polyline
- a width that specifies the interval between X coordinates in the point list (number of fullwords)
- the list of coordinates

The `width` parameter lets your application associate a variable number of array elements to each coordinate point. This allows application specific information to be included within the point list. The system ignores this information when constructing the polyline structure element.

Suppose your application uses the **GPPL2** subroutine, and the array contains X, Y, Z, and T coordinates, where T can be the temperature at location (x,y,z), and the elements arranged 1x,1y,1z,1t,2x,2y,2z,2t,...Nx,Ny,Nz,Nt. This presents no problem. Your application specifies the number of points, a width of four, and the name of the array.

By specifying a width of four, your application knows that the interval between X coordinates is four array elements. By virtue of specifying a two-dimensional subroutine, such as **GPPL2**, the system automatically reads only the first two coordinates (X and Y values) in each interval. In a two-dimensional case such as this one, the effect of this width specification causes the system to skip the third and fourth elements of each interval. For all two-dimensional subroutine calls, the Z value automatically defaults to 0.

A two-dimensional specification is simply a shorthand way of specifying that Z=0. The system treats all graphics data as three-dimensional. (In reality, many devices will optimize on two-dimensional data. For the purposes of understanding how the graPHIGS API operates, you should think of 2D data as 3D with Z=0.).

The following figure depicts one way to use the polyline output primitive to construct the parts of a cube. At each stage, the application adds another polyline to the model. Each new polyline is highlighted by a bold line.

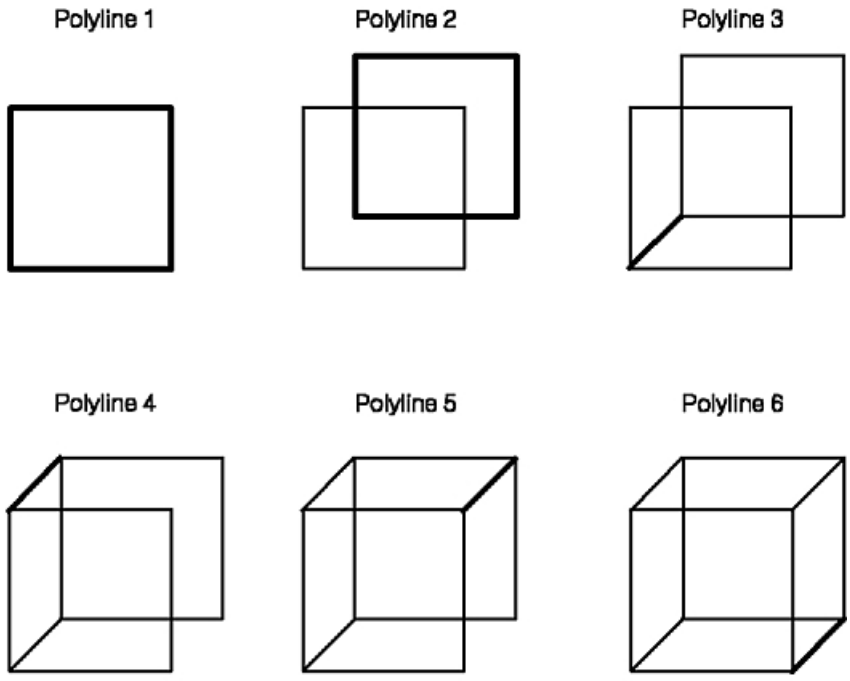


Figure 7. Sample Polyline. This image depicts six stages to construct the parts of a cube using the polyline output primitive. The first stage uses four polylines to draw a square that will be the front face of the cube. The second stage depicts a square, drawn with 4 polylines, pictured to the right and higher than the front face, to represent the back face of the cube. In third stage, a polyline is drawn connecting the lower left hand corner of the front face to the corresponding corner on the back face. In fourth stage, a polyline connecting the top left hand corner of front face to the corresponding corner on the back face is drawn. In the fifth stage, a polyline connecting the top right hand corner of the front face to the corresponding corner of the back face is drawn. In the final stage, a polyline is drawn from the bottom left hand corner of the front face to the corresponding corner of the back face to complete the cube.

The sample program you have coded so far to draw the house uses the polyline output primitive. The following is a completely different program. It also uses the polyline output primitive and will display a star. The coordinates of the star are coded in the variable `STAR` the program is coded using the FORTRAN language and will use the 6090 workstation for display.

Sample Program 2

Create this new sample program to learn how the graPHIGS API uses workstation default attribute values.

```
* DECLARE VARIABLES
      INTEGERx4 STATUS,CHOICE
      INTEGERx4 WSID,STRID,VIEW1
      REALx4 WINDOW(4),VIEWPT(4)
      REALx4 STAR(12)
      DATA WSID /1/
      DATA STRID /1/
      DATA VIEW1 /1/
      DATA WINDOW /-10.0,10.0,-10.0,10.0/
      DATA VIEWPT /0.0,1.0,0.0,1.0/
      DATA STAR /-6.0,2.0,6.0,2.0,-4.0,-6.0,
     *           0.0,6.0,4.0,-6.0,-6.0,2.0/
*********************************************************************
* OPEN FUNCTIONS
       CALL GPOPPH('SYSPRINT',0)
       CALL GPCRWS(WSID,1,7,'IBM5080','5080    ',0)
*********************************************************************
* VIEW DEFINITION
       CALL GPXVR(WSID,VIEW1,14,VIEWPT)
       CALL GPXVR(WSID,VIEW1,16,WINDOW)
       CALL GPXVCH(WSID,VIEW1,1,8,2)
*********************************************************************
* DATA CREATION
       CALL GPOPST(STRID)
       CALL GPPL2(6,2,STAR)
       CALL GPCLST
*********************************************************************
* DATA DISPLAY
       CALL GPASSW(WSID,1)
       CALL GPARV(WSID,VIEW1,STRID,1.0)
       CALL GPUPWS(WSID,2)
*********************************************************************
* INPUT FUNCTIONS
       CALL GPRQCH(WSID,1,STATUS,CHOICE)
*********************************************************************
* CLOSE FUNCTIONS
       CALL GPCLWS(WSID)
       CALL GPCLPH
       STOP
       END
```

The following figure shows the graphic display you should get when you run the program:

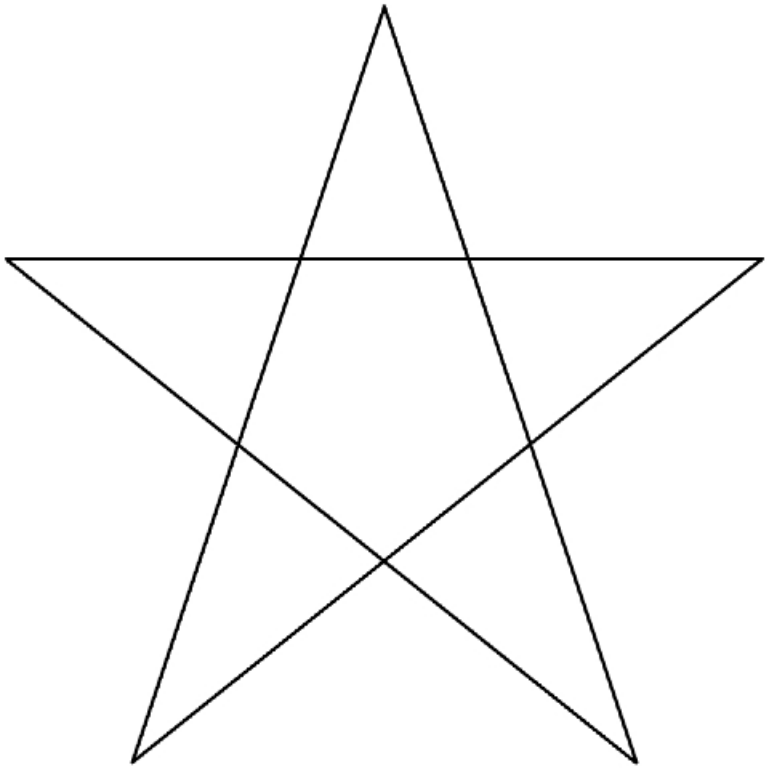


Figure 8. Output from Star Sample Program. This illustration shows a two-dimensional five point star.

Your drawing is a two-dimensional five point star. To draw a complete star the first and last coordinates must be the same. The total number of XY coordinates in the array is six. To display the star structure 1 contains only one element, a two-dimensional polyline structure element. No attribute elements are included in the structure, so the graPHIGS API uses workstation default attribute values. The star will appear white with solid lines.

Polyline Attributes

The graPHIGS API lets your application control the attributes of the polyline output primitive from individual settings, a bundle table, or a mixture of both.

Individual Polyline Attributes: The individually set polyline attributes include:

- Linetype
- Line Width Scale Factor
- Polyline Color
- Polyline End Type.

The Set Linetype **(GPLT)** subroutine lets the application program specify an index into the workstation's line type table. The default line type table contains the following entries:

- `1=SOLID_LINE` (system initialized value)
- `2=DASHED`
- `3=DOTTED`
- `4=DASH_DOT`
- `5=LONG_DASH`
- `6=DOUBLE_DOT`
- `7=DASH_DOUBLE_DOT`
- `8-`*n*`=SOLID_LINE`

The Set Linewidth Scale Factor **(GPLWSC)** subroutine lets the application program specify how wide to make the polyline. The application program specifies the width as a fraction of the workstation's nominal linewidth. For example, a value of 2 yields a line twice as thick, within the capabilities of the device.

The Set Polyline Color Index **(GPLCI)** subroutine lets your application specify a color by providing an index value that points to an entry in a workstation's color table.

The Set Polyline End Type **(GPPLET)** subroutine controls the appearance of line ends. Line ends may either be flat (default), round or square. Flat ends are the default and are placed at the endpoints of lines. Round ends are semicircles centered at the line end, having a diameter equal to the linewidth. Square ends are identical to flat ends except they extend one half linewidth past the line's endpoint.

Bundled Polyline Attributes: The API lets your application specify and use bundled polyline attributes.

The Set Extended Polyline Representation **(GPXPLR)** subroutine enables your application to load a polyline bundle entry, on a specified workstation, consisting of polyline line type, linewidth scale factor, and color values. These values are discussed above, individually.

Your application specifies which table entry to load through a parameter of the **GPXPLR** subroutine. This procedure is used to modify the workstation's polyline bundle table.

The Set Polyline Index **(GPPLI)** subroutine creates a structure element that points to an entry in the workstation's polyline bundle table. That entry contains a set of polyline attributes either set by the **GPXPLR** subroutine or set by default.

When encountering a polyline primitive at draw time, the system applies only those bundled attributes that are in effect at that time, as specified by the polyline ASFs.

Modified Sample Program 2

Refer back to the star Sample Program. Modify it according to the following:

1. Add the following statements to the `DECLARE VARIABLES` section:

```
INTEGERx4 ASFID(2),ASFLAG(2),COLOR(2)
DATA ATLIST /1,3/
DATA ATFLAG /1,1/
DATA COLOR /1,2/
```

2. In the `DATA CREATION` section, add the following statement before the `GPOPST(STRID)` statement:

```
CALL GPXPLR(WSID,4,1,2)
CALL GPXPLR(WSID,4,2,1.0)
CALL GPXPLR(WSID,4,3,COLOR)
```

3. In the same section, replace structure 1 with the following statements:

```
CALL GPOPST(STRID)
CALL GPASF(2,ASFID,ASFLAG)
CALL GPPLI(4)
CALL GPLWSC(2.0)
CALL GPPL2(6,2,STAR)
CALL GPCLST
```

Recompile, load, and run the program after inputting these modifications. The following figure shows the image displayed:

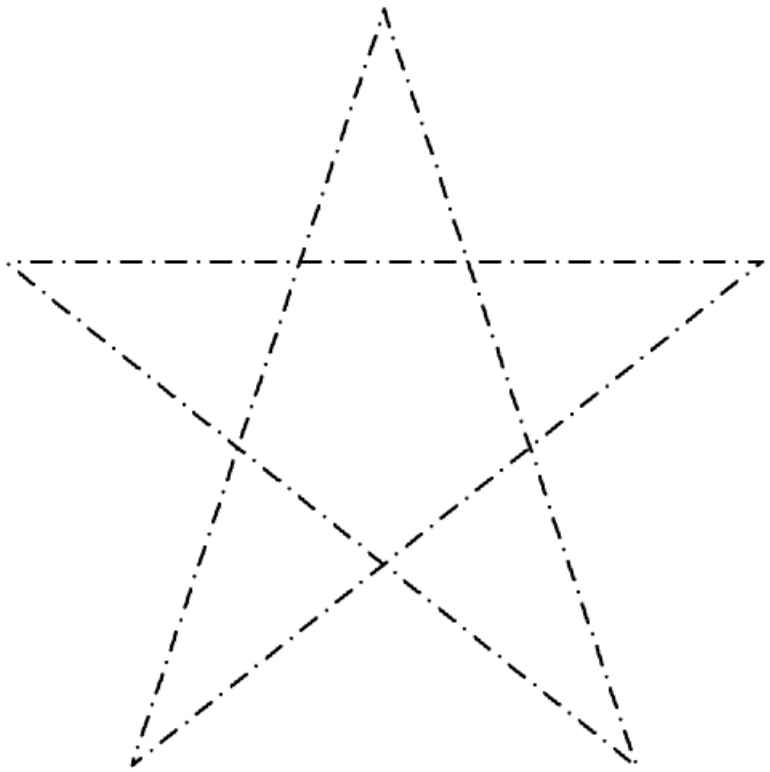


Figure 9. Polyline Star with New Attributes. This illustration shows a two-dimensional five point star with a dash-dot-dash line type.

The changes explained above will allow your application to use mixed attribute specifications. By using **GPXPLR** you have changed polyline bundle table index four. In the structure, the **GPASF** structure element specifies that only the line type and color attributes are bundled. Then, the **GPPLI** indicates the bundle table index for these attributes. The **GPLWSC** specifies the line width scale factor individual attribute. The star is drawn with red dashed lines.

Polymarker

Polymarkers give your application the ability to specify transformable point locations using various types, sizes, and colors of markers.

A single polymarker output primitive can mark several points. The following figure depicts various polymarker output primitives. The squares are not part of the polymarker output primitive; they are included in the drawing for explanation purposes only.

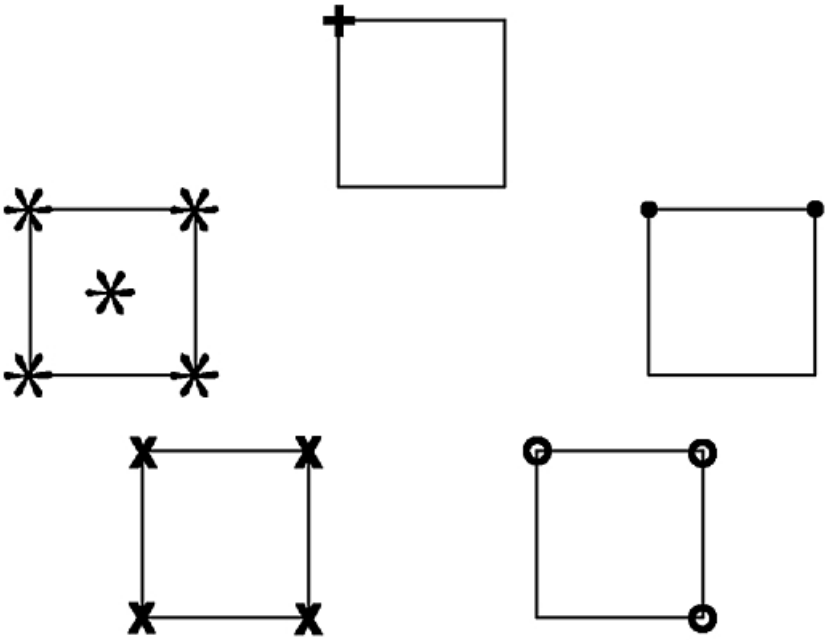


Figure 10. Marker Type Illustrations. This illustration shows five two-dimensional squares. Each square is marked with a different polymarker output primitive: plus sign, solid circle, open circle, the letter x, and asterisk.

Polymarker Output Primitive

If you need to mark points, use the Polymarker **(GPPM2** and **GPPM3**) subroutines. These subroutine calls create structure elements that are used to specify markers in either two- or three-dimensional Modeling Coordinate Systems (MCS).

As with the polyline and polygon primitives, your application provides a series of modeling coordinate points. Your application must declare a pointlist with X, Y, and optionally Z coordinates to locate the desired number of polymarkers.

Only the position of markers is fully transformable. In order to change a marker's size, the API provides a scale factor that operates independently of any scaling done to the model.

Chapter 5. Viewing Capabilities

Using the graPHIGS API programming tools enables your application to assemble a graphics model. The assembled graphics model is defined in a system of coordinates known as the World Coordinate System (WCS). The viewing subroutine calls provided by the graPHIGS API enable your application to define what part of the WCS is visible. They also help to simulate various camera-like operations such as flying through space, panning, and zooming.

In order to establish what part of the WCS is visible, your application must define a view. Defining a view involves specifying its:

- Orientation through a view matrix
- Clipping volume (window and near/far extents that specify what is within the viewer's field of vision)
- Priority relative to other views
- Characteristics
- Mapping to an output device

The graPHIGS API lets your application define and modify viewing parameters. Calling a viewing subroutine sets the requested viewing entries in a workstation's view table. When a workstation is updated, the requested values become the current values. Updating the workstation is discussed in Chapter 6. Displaying Structures.

Views are stored in a view table as part of the WSL. The number of entries in the view table may vary for each workstation and is defined in the WDT. All the view table entries of the WSL are initialized to default values. These values are explained in the *The graPHIGS Programming Interface: Technical Reference*. Your application may modify any entry in a view table except view zero.

In the sample program introduced in Chapter 1, the following statements defined the viewing parameters for view index 1 of the workstation view table:

```
CALL GPXVR(WSID,VIEW1,14,VIEWPT)
CALL GPXVR(WSID,VIEW1,16,WINDOW)
CALL GPXVCH(WSID,VIEW1,1,8,2)
```

The first two statements define the window and viewport and the next statement the view characteristics. The view matrix used is the default matrix. The priority of this view is not modified.

The following sections discuss ways to modify viewing parameters and the effects of modifying entries in a workstation's view table.

View Orientation Information

In order to establish a view, other than the system's predefined view, you must specify a viewing transformation matrix. This matrix establishes a Viewing Coordinate System (VCS).

Initially, the system's predefined view places the VCS coincident with the WCS. The following figure depicts the relationship between the World Coordinate System and the Viewing Coordinate System:

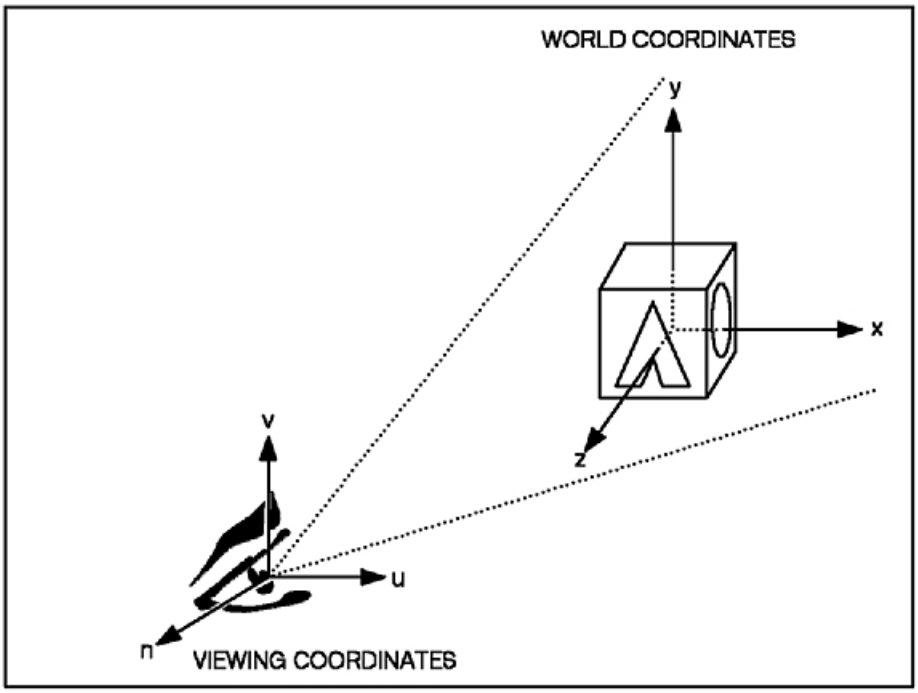


Figure 11. Relationship between the WCS and the VCS. This illustration shows an eye looking at a box. The eye is in viewing coordinates (u,v,n); the box is in world coordinates (x,y,z).

The Viewing Coordinate System serves as a frame of reference for the viewer. With this frame of reference, your application can define the viewer's field of vision as discussed in Window and Viewport Definition.

Your application can use the default viewing transformation matrix. The default establishes the origin of the VCS at the origin of the WCS, and aligns the N and V axes in the VCS with the Z and Y axes in the WCS, respectively.

The following figure depicts how the viewer's field of vision follows changes made to the VCS:

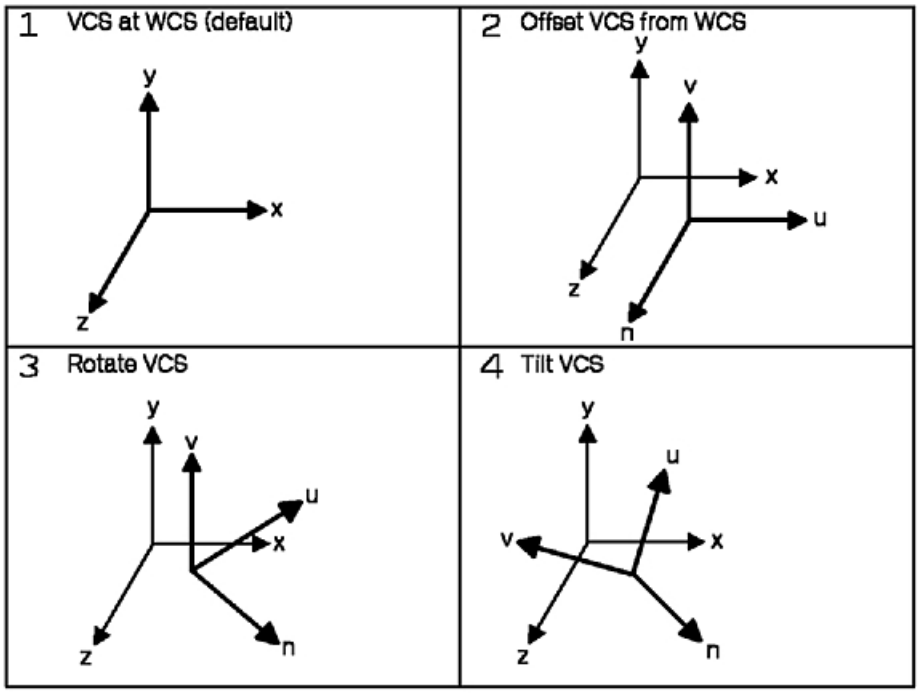


Figure 12. How the Viewer's Field of Vision Follows the Changes Made to the VCS. Four illustrations depict cumulative changes in the VCS against a static WCS. The first illustration shows the VCS axes equivalent to the WCS (default); the second shows them offset from the WCS; the third shows them rotated about the v-axis; and the fourth shows them tilted around the n-axis.

The Set Extended View Representation **(GPXVR)** subroutine enables your application to set different fields in the workstation view table entry.

By setting specific parameters of the **GPXVR** subroutine, your application can assign two- or three-dimensional transformation matrixes to entries in a workstation's view table. The definition of the viewing matrix is applied when the API displays the root structures associated with that view.

Other aspects of defining a view are discussed in the following sections of this chapter. Displaying a view and its contents is discussed in Chapter 6. Displaying Structures.

Window and Viewport Definition

This section discusses the programming subroutine calls that enable your application to define:

- what is visible to a virtual viewer, that is, what part of the World Coordinate System is within the *clipping boundaries* that define the viewer's field of vision
- whether the type of viewing projection is *parallel* or *perspective*
- how a virtual viewer's field of vision is *mapped to a logical viewport*

The Set Extended View Representation **(GPXVR)** subroutine will also give your application the ability to provide the above viewport and window definitions for two- or three-dimensional viewing.

The following sections discuss how each aspect of the window and viewport definition contributes to the viewing environment.

Notice that your application has the option of using the system's default values for window boundary, type of viewing, and viewport mapping specifications. Refer to *The graPHIGS Programming Interface: Subroutine Reference* for further details.

Clipping Boundaries

The graPHIGS API provides parameters in the **GPXVR** subroutine that let your application define the clipping boundaries of a two- or three-dimensional field of vision. In order to define clipping boundaries, your application specifies, within the VCS:

- The distance of the view plane relative to the view reference point
- The boundaries of the window (defined on the view plane)
- The near and far clipping planes

These parameters let your application define what part of the WCS is visible to the viewer. Typically, anything that is not within this viewing volume is not seen. Models or parts of models that are outside the viewing volume may be optionally clipped (excluded from view), and therefore not displayed.

The following figures depict how changes to the viewing volume can zoom in on and clip the front or back of the model. Notice that the projection type in these figures is parallel and that the viewport remains consistent throughout.

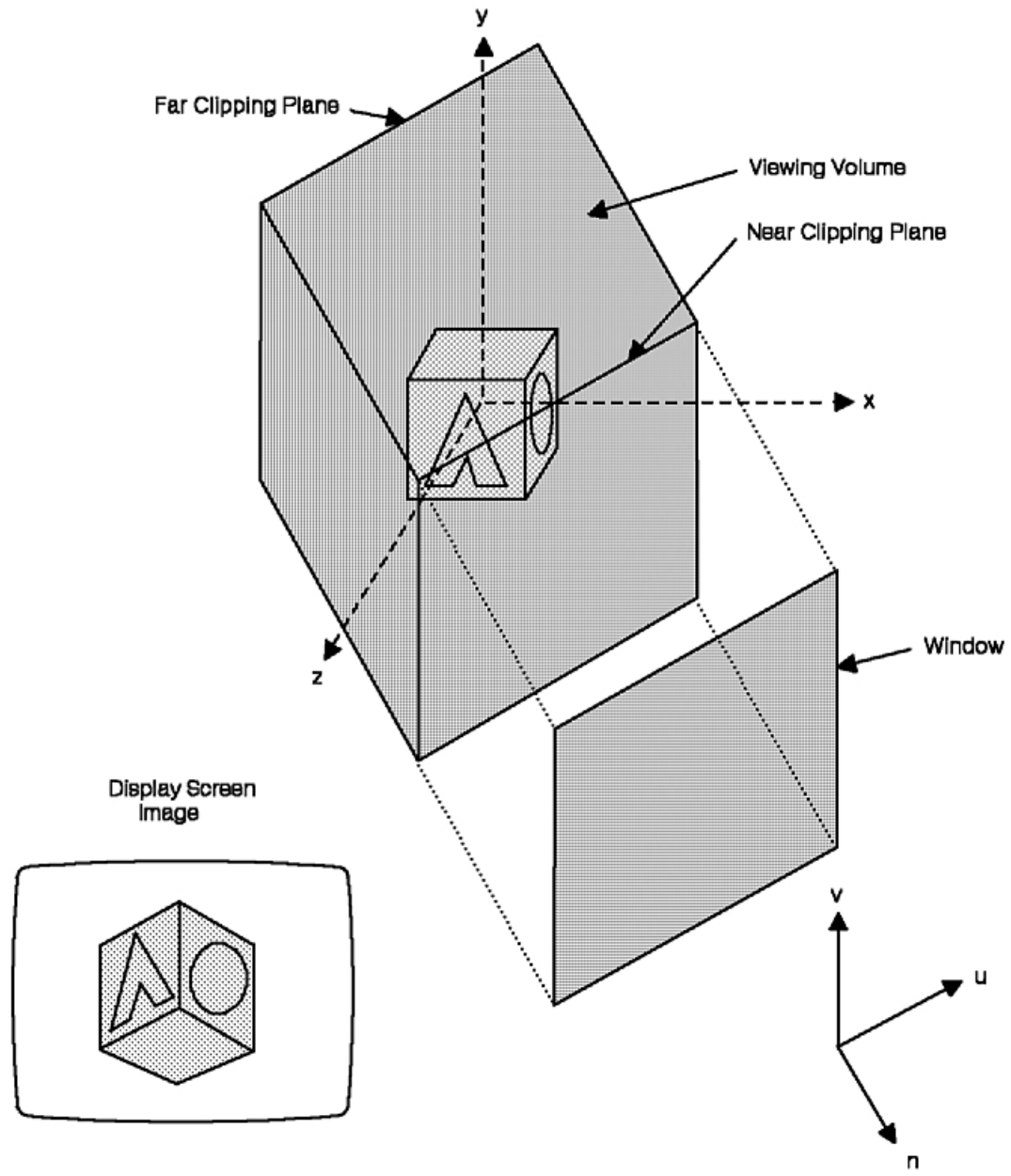


Figure 13. Modifying the Viewing Volume (Part 1 of 4). Initial Viewing Volume Specification. This illustration shows the initial viewing volume surrounding an object. The near clipping plane lies between the eye and the object, with the far clipping plane behind. An inset shows the final display screen image; it is unclipped.

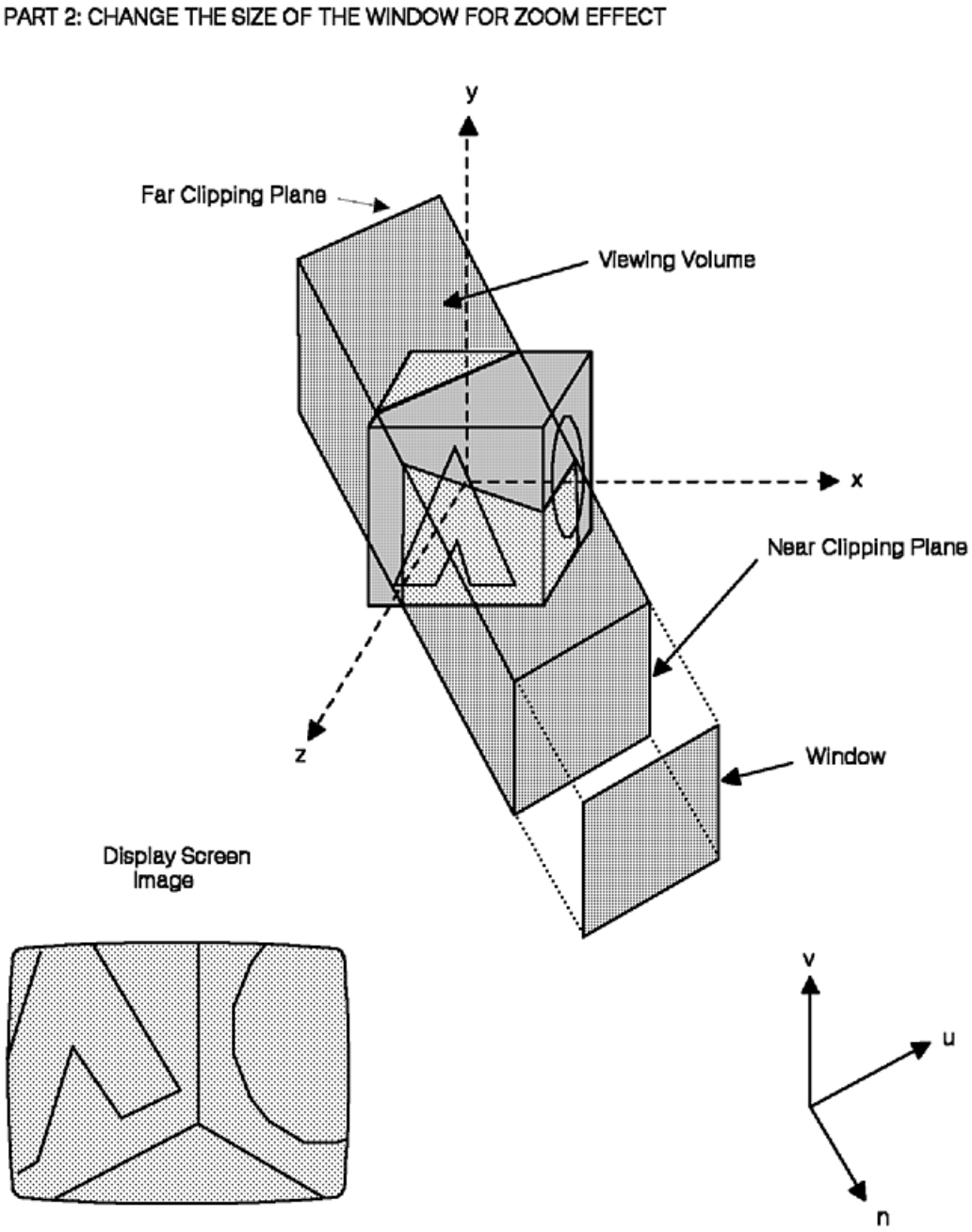


Figure 14. Modifying the Viewing Volume (Part 2 of 4). Change the Size of the Window for Zoom Effect. This illustration shows a smaller window size; the view volume is now too small to contain the entire object. The final display shows the object zoomed and clipped to the window.

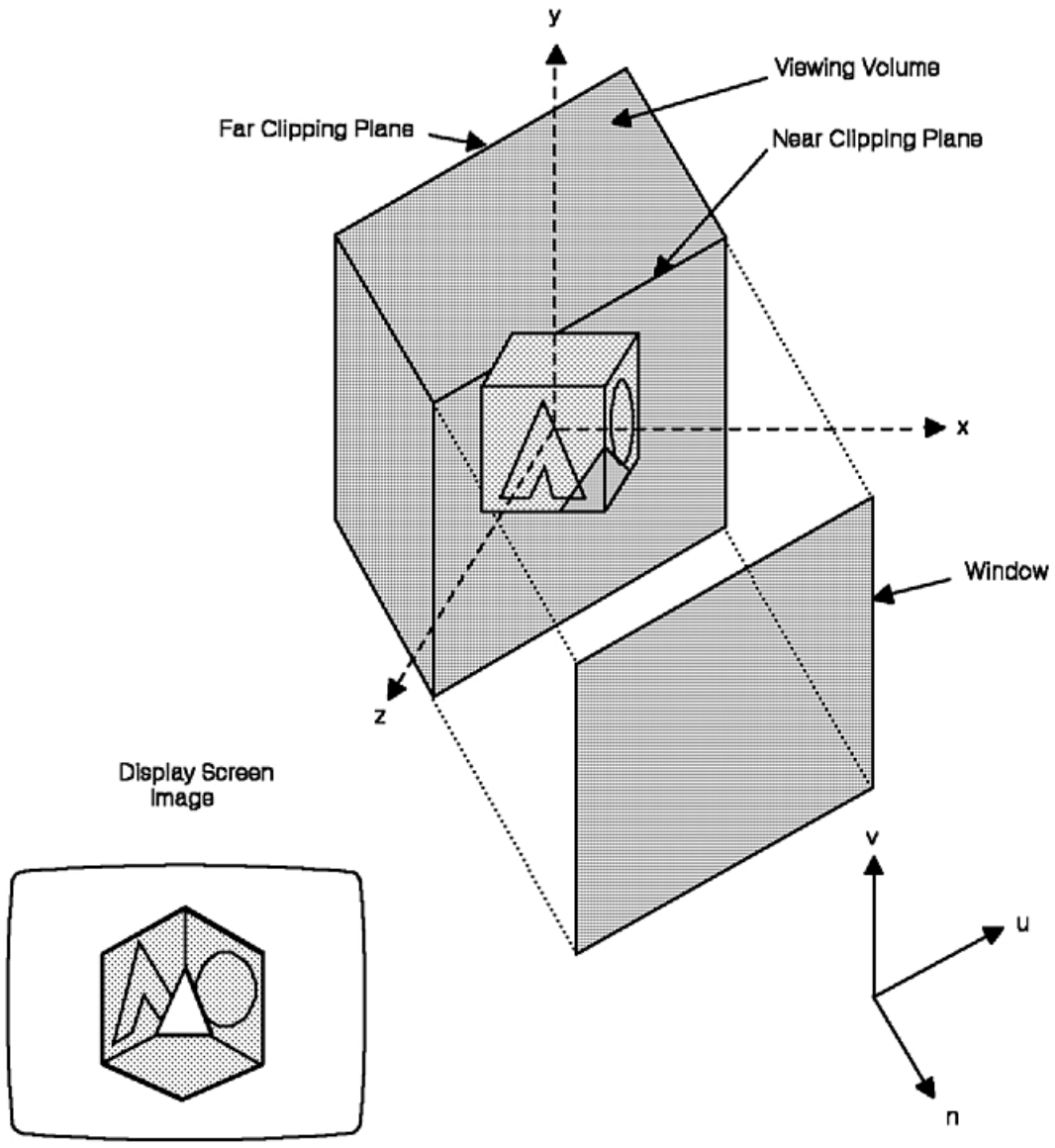


Figure 15. Modifying the Viewing Volume (Part 3 of 4). Enlarge Window and Move Near Clipping Plane. This illustration restores the initial window size, but moves the near clipping plane back until it clips the front of the object. The final display shows the object with the front edge trimmed away.

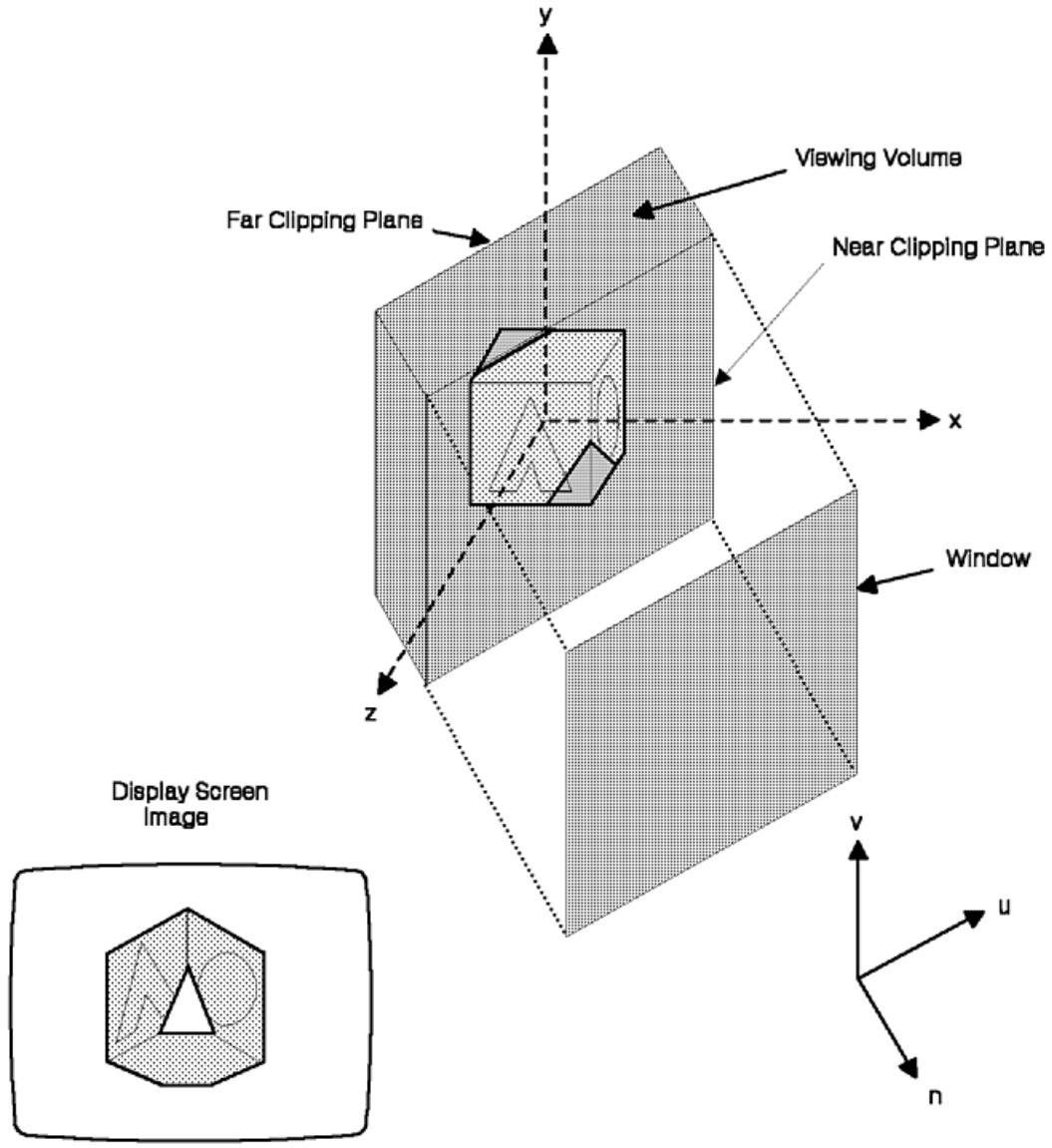


Figure 16. Modifying the Viewing Volume (Part 4 of 4). Move Far Clipping Plane. This illustration continues by moving the far clipping plane forward until it clips the back of the object. The final display shows the object with both the front and back edges trimmed away.

Parallel and Perspective Viewing

For three-dimensional viewing the graPHIGS API provides a parameter in the **GPXVR** subroutine that enables your application to request two viewing orientations:

- `1=PARALLEL`
- `2=PERSPECTIVE`

The following figure depicts how parallel and perspective viewing affect the viewing volume and the image of a cube.

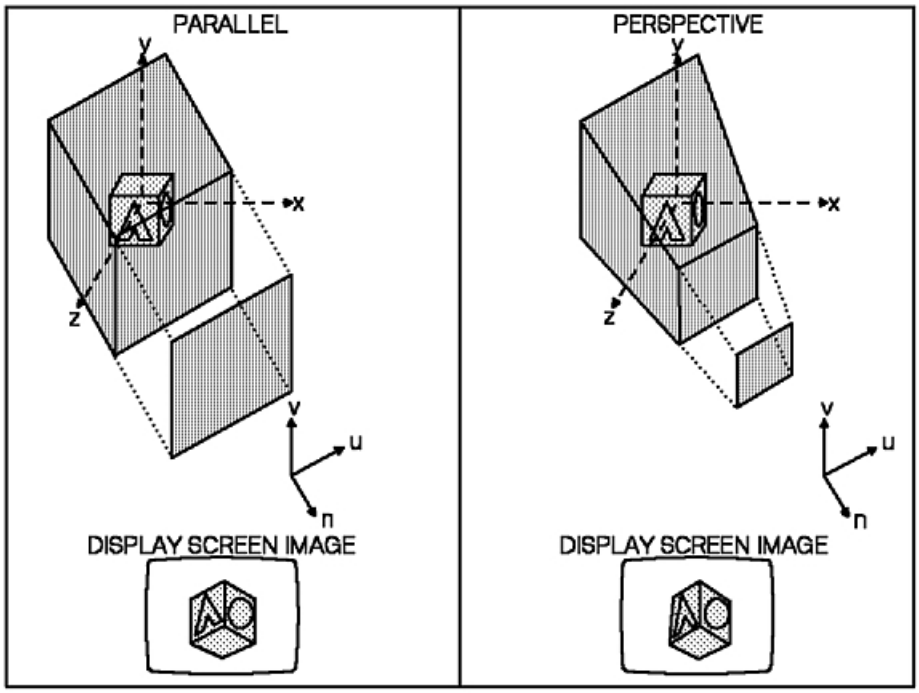


Figure 17. How Parallel and Perspective Viewing Affects Viewing Volumes and Images. Two illustrations show the difference between parallel and perspective views. The parallel illustration shows a cubic view volume and a display screen image that looks "squashed," as though the object were flattened into the screen. The perspective illustration shows a view frustum that is narrower towards the eye and wider away from the eye; the resulting display screen image has a more natural appearance.

The following figures depict three side-views of typical window, view reference point, and projection reference point placements. The projection reference point is used in conjunction with the center of the window to define the direction of projection. For `PARALLEL` projection, all projectors are parallel to the line defined by the projection reference point and the center of the window. For `PERSPECTIVE` projection, all projectors converge on the projection reference point. The viewer's field of vision is defined by those converging lines that pass through the window and are within the near and far clipping planes. The first part shows `PARALLEL` projection. The second part shows `PERSPECTIVE` projection. Each part also shows the resulting display screen image.

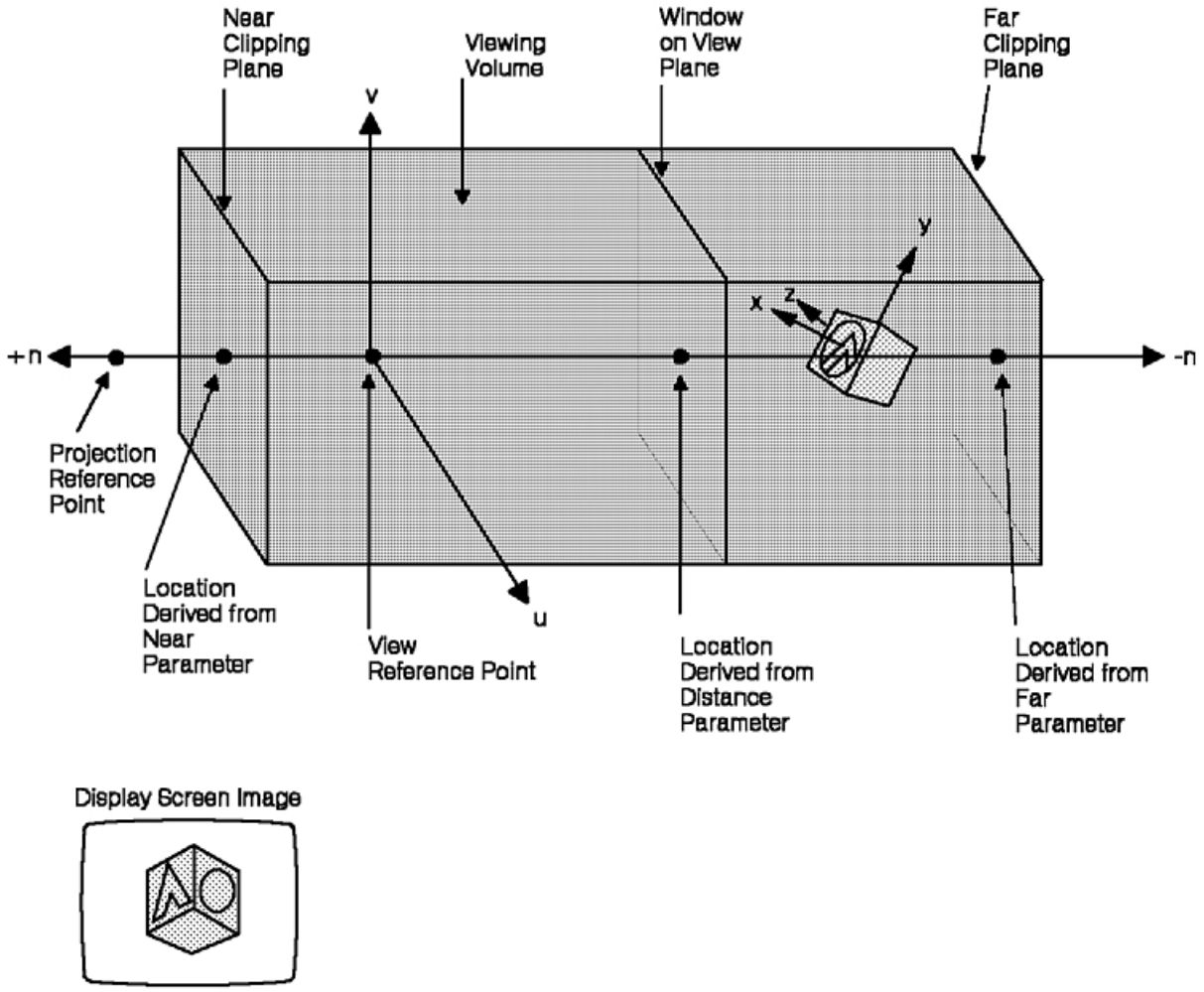


Figure 18. Sample Placements of View Mapping Parameters (Part 1 of 3). Side View of Normal Parallel Projection. This illustration is similar to the first half of figure above, but shown from the side. The Projection Reference Point is shown behind the eye, in the "+n" direction. The Near Clipping Plane is shown next, still behind the eye, followed by the View Reference Point (the eye itself). The View Plane, World Coordinate object, and Far Clipping plane follow in the "-n" direction. The Viewing Volume is rectangular and the display screen image looks squashed into the screen.

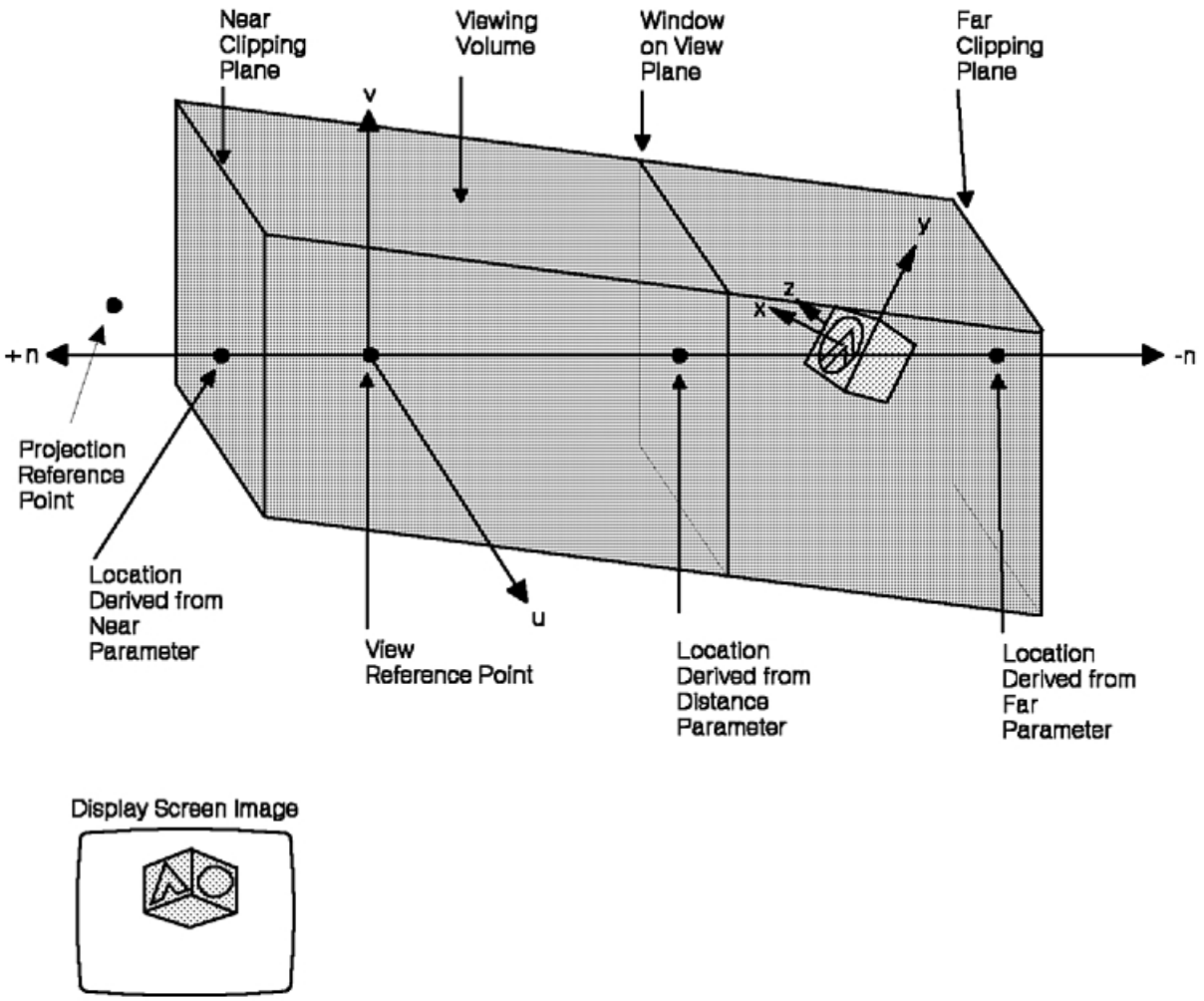


Figure 19. Sample Placements of View Mapping Parameters (Part 2 of 3). Side View of Oblique Parallel Projection. This illustration is identical to the previous, but the Viewing Volume is sheared into a rectangular prism with the far plane lower than the near plane. The resulting display screen image looks like the normal parallel projection, but is also squashed vertically.

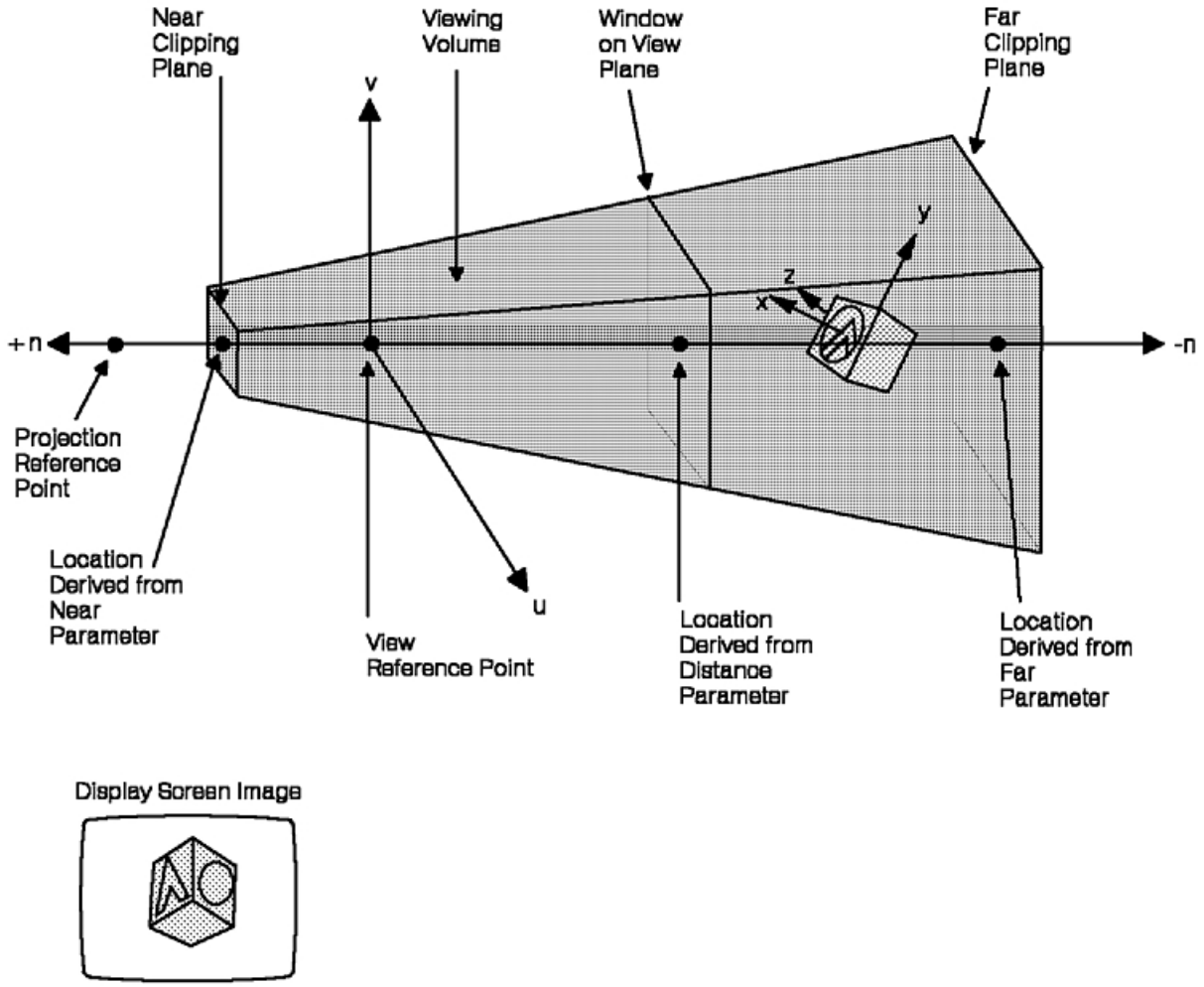


Figure 20. Sample Placements of View Mapping Parameters (Part 3 of 3). Side View of Perspective Projection. This time the Projection Reference Point is used to squash the near plane, turning the Viewing Volume into a frustum. This illustration and its resulting display screen image are similar to the second half of the previous figure, How Parallel and Perspective Viewing Affects Viewing Volumes and Images, but shown from the side.

Viewport Mapping

The graPHIGS API provides a parameter in the **GPXVR** subroutine that lets your application control where to position the view. The system maps the view to a viewport in a logical workstation space called *normalized projection coordinates* (NPC). NPC is defined as a cubic space ranging from zero to one along each axis.

Your application specifies the boundaries of a two- or three-dimensional viewport in NPC. This viewport serves as a destination for the view in NPC.

NPC is a workstation-dependent, device-independent space in which to specify a logical viewport. Creating a logical viewport gives different devices a common set of values from which to base a mapping to their actual screen.

The next mapping, by default, maps all of NPC onto the entire workstation viewport. This mapping may be modified and is discussed in Mapping of NPC to Output Devices. The following figure depicts the stages involved in the viewing process from the creation of a VCS in the WCS to the viewport mapping into NPC:

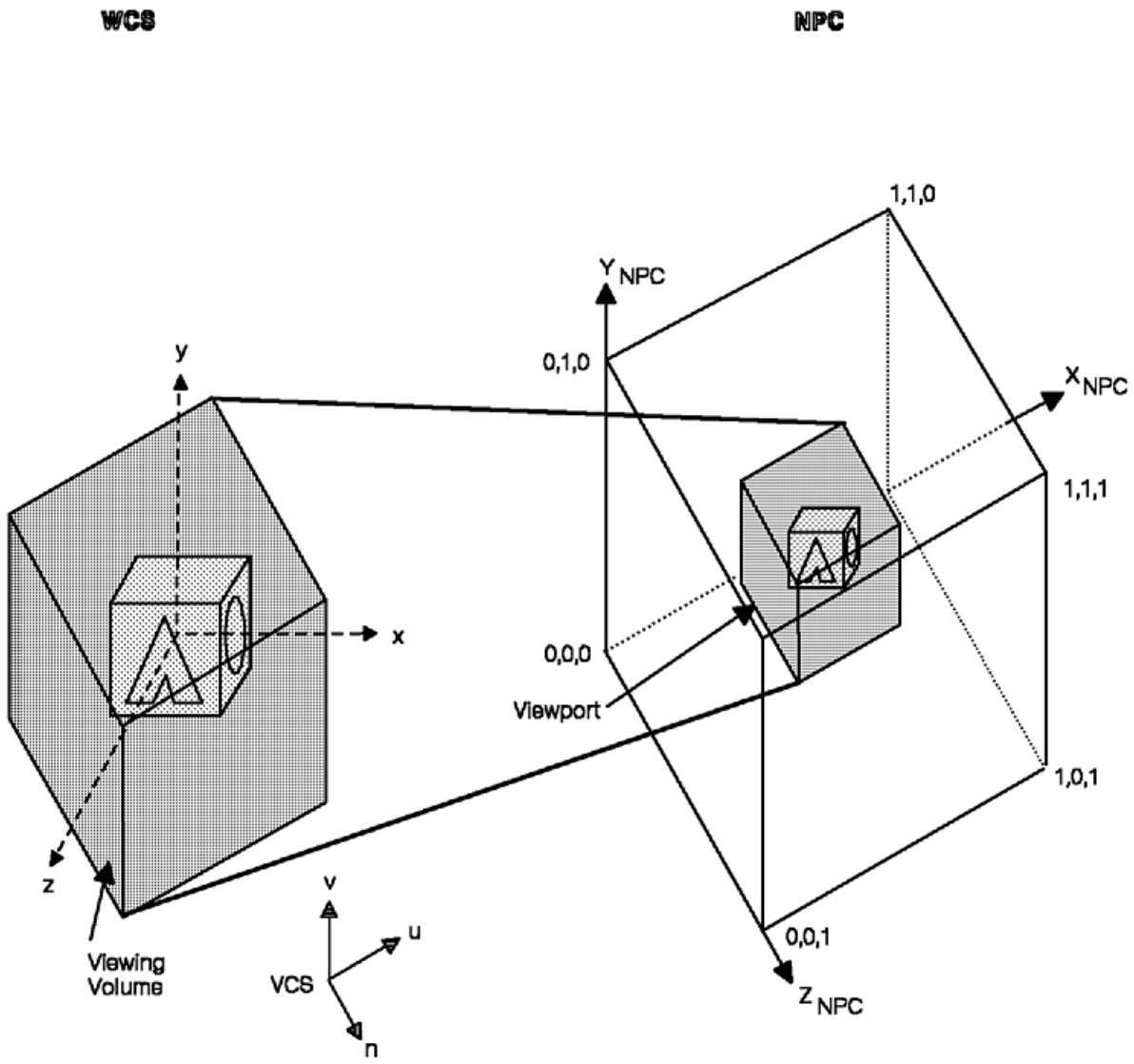


Figure 21. Staged Representation of Viewing Process. This illustration depicts a viewing volume and how it gets mapped into a viewport. The viewing volume is specified in WCS as a cubic space. This volume gets mapped into a viewport within the application designated NPC, which ranges from zero to one along each axis.

A one-to-one correspondence exists between a view and a viewport. The following two sections: View Priority and View Characteristics show you how to control the appearance of viewports and their relation to other viewports on a display surface.

View Priority

The Set View Output Priority **(GPVOP)** subroutine lets your application specify a view's output priority relative to another view. This determines the order in which the system presents each view on the output device. Higher priority views are drawn last so they overlay the lower priority views.

The Set View Input Priority **(GPVIP)** subroutine is similar to **GPVOP**, but sets a view's relative input priority for `LOCATOR`, `STROKE`, and extended `PICK` input. To set the view input and view output priorities at the same time, your application should issue a Set View Priority **(GPVP)** subroutine. Mixing different view input and output priorities is discussed in detail in Chapter 15. Advanced Viewing Capabilities in Part Two of this book.

When a workstation is opened, its view zero has the highest input and output priority. Notice that this ordering is not static. The graPHIGS API supports dynamic modification of view priorities.

The following figure depicts various changes to the view output priorities of fifteen viewports. In the figure, the application gave view 3 a higher priority than view 1 (which has a higher output priority than view 2). Also, the application gave view 9 a lower output priority than view 10.

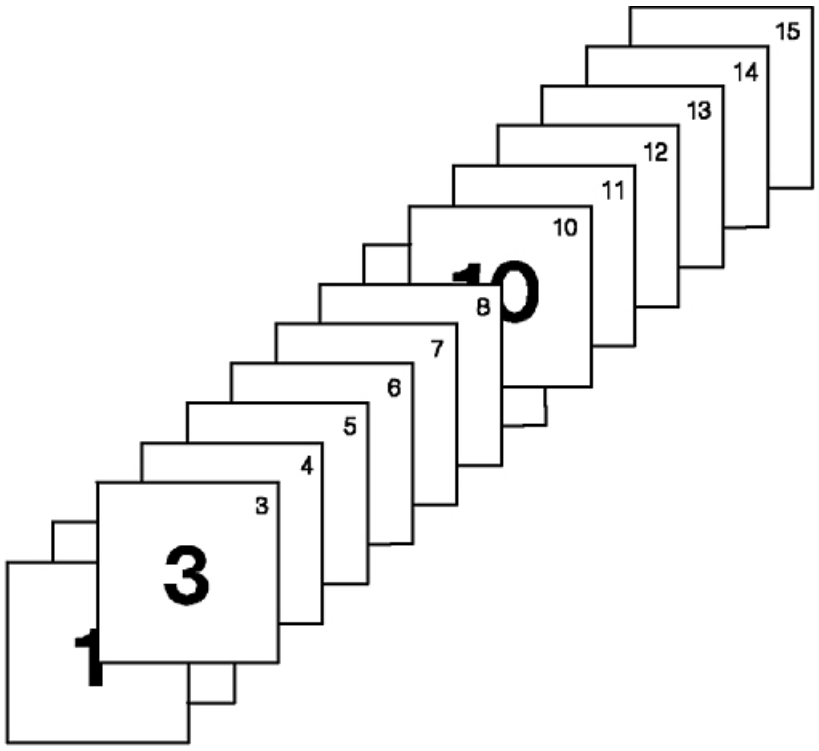


Figure 22. Controlling View Priorities. This illustration fifteen viewports, each depicted as squares, with priority that corresponds to the view number; view 1 has a higher priority than view 15. Two exceptions to the priority list are depicted in this illustration: view 3 has a higher priority than views 1 and 2, and view 10 has a higher priority that view 9 but lower than view 8. Therefore, view 3 overlays views 1 and 2 and has the highest priority. View 10 overlays view 9, but both view 9 and 10 are overlayed by view 8.

View Characteristics

The graPHIGS API gives your application control over a viewport's appearance on a display. The Set Extended View Characteristics **(GPXVCH)** subroutine lets your application:

- activate and deactivate views
- activate and deactivate the clipping boundaries

- control the shielding and shielding color of each view
- control the visibility and color of each view's border
- set temporary view indicator

You should know that the graPHIGS API does not process the content associated with inactive views, although the view's content is still defined to the workstation. The shielding characteristic controls the transparency of the view. When the Shielding Indicator is ON the view background is filled and it shields the content of all lower priority views behind the extents of that viewport. The following figure depicts how these controls affect the appearance of views:

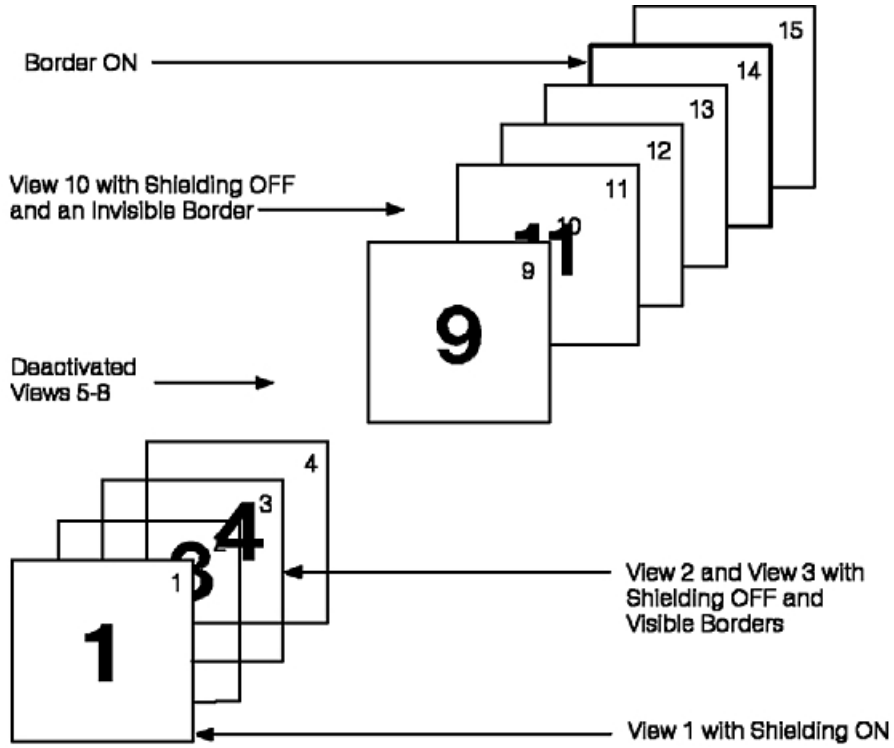


Figure 23. Controlling View Characteristics. This illustration depicts fifteen views as in the previous figure. View 1 has the highest priority and view 15 has the lowest with no exceptions. This illustration depicts various view characteristics to show the effect on vies. View 1 has shielding on so the view background is displayed and all lower priority views are occluded behind the extents of this view. Views 2 and 3 have shielding off and visible borders, therefore view 4 is visible through views 2 and 3. Views 5 through 8 have been deactivated so they are not displayed. View 10 is displayed with shielding off and an invisible border so view 11 is visible and not occluded by view 10. View 14 has the border on, so the view has an outline outside of the view background (shielding).

Window clipping, near clipping, and far clipping may each individually be turned ON or OFF. When your application disables any of these clipping boundaries, the API performs clipping to the workstation which is described in the next section. This remains in effect unless your application modifies the relationship between NPC space and the display surface as part of the workstation transformation discussed in Mapping of NPC to Output Devices. Also, if window clipping is turned OFF, the data in that view may extend beyond its corresponding viewport boundaries, and may overlap data in other views. Nevertheless, in this case, the same viewport shielding remains in effect.

When set to ON, the temporary view indicator identifies to the workstation support software that the corresponding view will have its view activity changed frequently. When the corresponding temporary view is activated the image of the underlying screen is saved by the device. When it is deactivated the underlying screen image is restored without requiring structure traversal. A typical application of the temporary view capability is the "pop-up-menu".

Extended view characteristics associated with advanced viewing concepts are discussed in detail in Chapter 15. Advanced Viewing Capabilities.

Mapping of NPC to Output Devices

The previous sections explained how to create logical viewports in NPC space. This section explains how to control the mapping of NPC space to the physical device.

The process of mapping NPC space to a device is called the *workstation transformation.* The Set Workstation Transformation **(GPWSX2** and **GPWSX3)** subroutine enable your application to specify the workstation transformation. This isotropic mapping is device-dependent and results in a representation of the model in Device Coordinate (DC) space.

By default, the system provides the most trivial and useful mapping. It maps the entire NPC space onto the largest usable portion of the workstation's DC space.

For example, your application might choose to map the entire NPC space (0,0,0 to 1,1,1 in the window parameter of the **GPWSX3** subroutine) to the entire display screen of an output device such as in the IBM 5080 Graphics System with a DC space from 0 meters to .28448 meters.

For devices with a square screen, this results in the mapping of all NPC onto the entire display surface. Typically, applications do not need to modify this relationship. One reason to modify the workstation transformation is to utilize the entire display surface of a workstation with a rectangular display surface, such as an IBM GDDM device.

Your application can determine the extents of a workstation's DC space using the Inquire Maximum Display Surface Size **(GPQDS)** subroutine. See the discussion of workstation related inquiries in Chapter 9. Inquiry Subroutines for more information on the use of inquiries.

The graPHIGS API also supports more advanced uses of the workstation transformation. Your application can choose to map any part of the NPC space **(GPWSX2** or **GPWSX3** workstation window parameter) to any supported part of DC space (workstation viewport) on the particular output device. Notice that during a workstation transformation the API maintains the same aspect ratio in the workstation viewport that existed in the workstation window, as shown in the following figure:

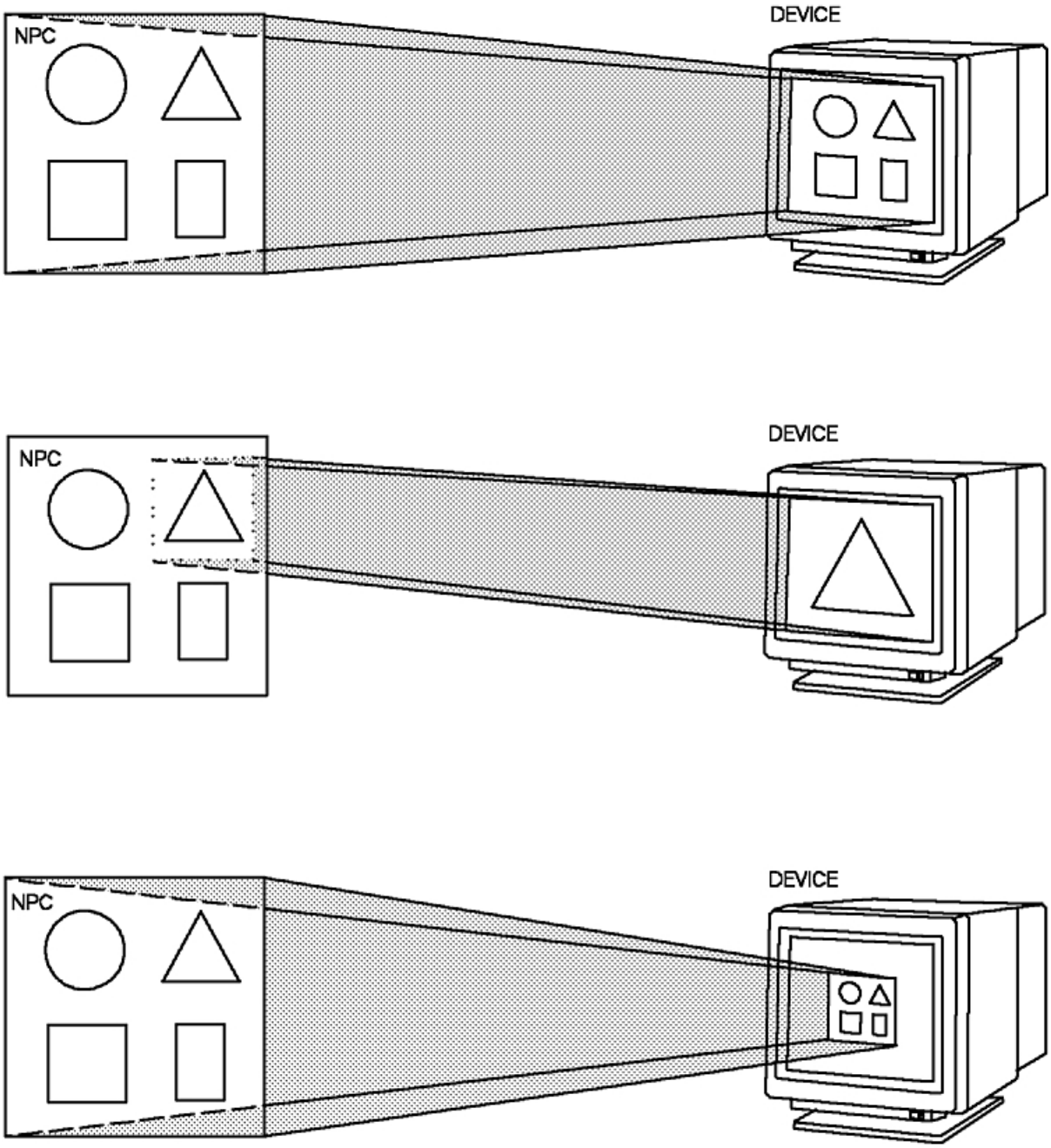


Figure 24. The Result of Various Workstation Transformations. This illustration demonstrates that the application can map all or part of the NPC space onto part or all of the device (workstation viewport).

Modified Sample Program 1

Make the following changes to the house program for a better understanding of view mapping. (See Sample Program and Sample Program 2.) You will add a new view definition and modify the old one.

- In the DECLARE VARIABLES section:
  - Add the following statements:

```
INTEGER*4 CHARID(2),VALUE(2)
REAL*4 WNDOW2(4),VIEWPA(4)
INTEGER*4 VIEW2
```

  - Modify the VIEWPT DATA statement to the following:

```
DATA VIEWPT /0.0,0.5,0.0,0.5/
```

  - Add the following statements:

```
DATA CHARID /6,8/
DATA VALUE /2,2/
DATA VIEW2 /2/
DATA WNDOW2 /-50.0,50.0,-50.0,50.0/
DATA VIEWPA /0.5,1.0,0.5,1.0/
```

- In the VIEW DEFINITION section:
  - Add the following statements:

```
GPXVR(WSID,VIEW2,14,VIEWPA)
GPXVR(WSID,VIEW2,16,WNDOW2)
GPXVCH(WSID,VIEW2,2,CHARID,VALUE)
```

- In the DATA DISPLAY section:
  - Add the following statement:

```
GPARV(WSID,VIEW2,STRID(1),1.0)
```

Make the changes in your sample program, compile it, and run it. The display should show two views of the house, as shown in the following figure. By introducing the changes explained, you have modified the view parameters for view index 1 and 2 of the workstation view table.

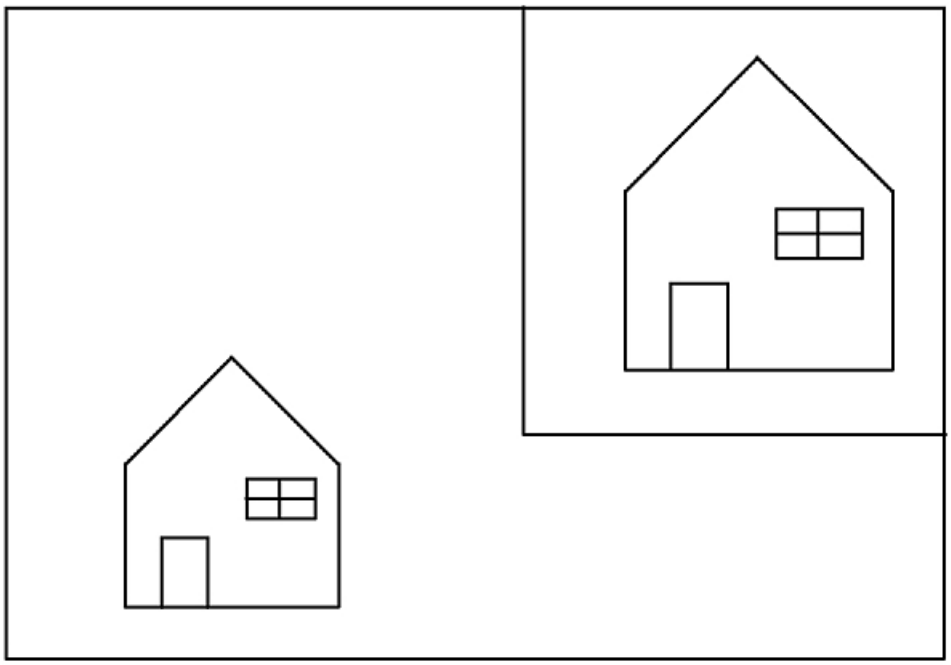


Figure 25. Two views of the house. This illustration shows two views of a house. View 1 has a larger window size than view 2, therefore the house in view 1 is smaller. View 2 has the border indicator on while view 1 has no border or shielding.

View 2 is located in the top right-hand corner, and view 1 in the lower left-hand corner. The same structure is associated to both views. You will learn about association of structures to views in the next chapter.

The difference in size of the houses in each view is due to the window size used. This is a zoom effect. View 1 has a larger window size than view 2, therefore the house in view 1 is smaller. Other methods for zoom effects use the view transformation matrix.

View 2 has the border indicator on, and the border color is white, no shielding is in effect. View 1 has no border and no shielding. Priorities have not been modified. The view matrix used is the default view matrix.

Chapter 6. Displaying Structures

Structures are defined in the centralized structure store, independent of any workstation. Your application has the following two types of facilities to display the graphical data in structures:

- Your application can associate a structure store to a workstation, and then associate structure networks, contained in the structure store, to views of the workstation. Your application can then update the workstation to initiate the process of generating the picture on the display.
- Your application can use Explicit Traversal Controls to cause direct generation of the picture by the workstation, bypassing the associating of structures with view and the update process. See Chapter 14. Explicit Traversal Control for details of the explicit traversal process.

This chapter discusses the functions that give your application control over whether its graphic data is associated with or disassociated from views and workstations and the way the graPHIGS API processes a root's structure network. It is essential that you understand this process in order to conceptualize how to achieve desired results on a display device.

Associating Structure Networks with Views and Workstations

This section discusses how to associate portions of graphics data with a workstation or workstations, and make it eligible for display. When your application associates a structure with a workstation or a view, that structure becomes known as a **root structure.**

In the sample program introduced in Chapter 1, the statements:

```
CALL GPASSW(WSID,1)
CALL GPARV(WSID,VIEW1,STRID(1),1.0)
```

first associates structure store 1 to all views of the workstation, then structure 1 (the root structure) is associated with view 1 of the workstation enabling your application to display the house on your graphics terminal. A detailed discussion of structure stores is contained in Part Two of this manual.

The graPHIGS API splits the process of associating roots into two program subroutine calls that let your application separately associate a root with a workstation and with a view. This lets your application take advantage of intelligent workstations that can store graphics data even when it is not needed for display. For example, your application may choose to store many menus at a workstation of which only one is displayed at a time. Another menu can then be displayed quickly by associating the structure to a view, in which case only the association data (and not the entire menu structure) needs to be sent to the workstation. The following figure depicts a workstation's storage in which stored structures are both associated and not associated with views:

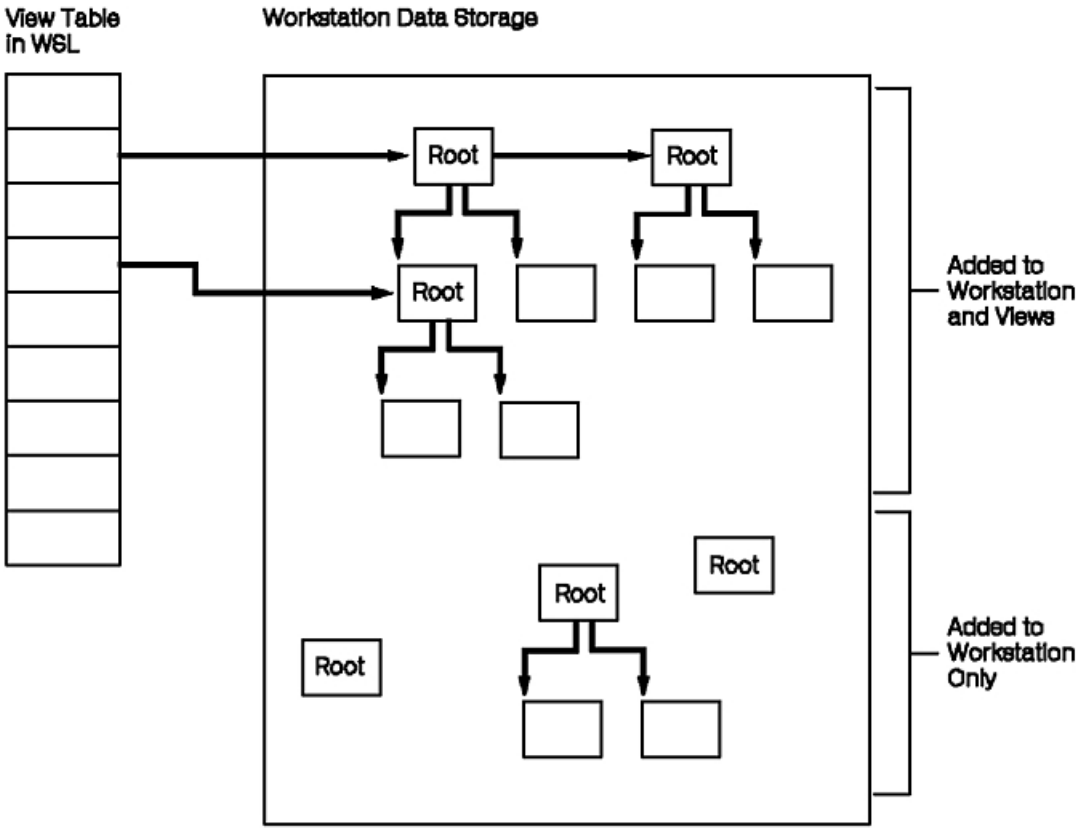


Figure 26. Workstation Capable of Storing Associated and Non-Associated Structures. This illustration shows that root structures can be associated to both a workstation and a view or to just a workstation. In this way, the workstation data is available to the workstation even if the data is not currently being displayed. Once the data is associated to the view as well as the workstation, then the graphics data will be displayed.

Associate Root with a Workstation

The Associate Root with a Workstation **(GPARW)** subroutine lets your application associate a structure network with a workstation. This lets your application pre-load structures that may be required later.

When the data resides at the workstation, your application only needs to associate these structures to a view for display. This reduces communication time, and provides better response time, since at the display time, only the root identification information is sent, not the graphics data.

Associate Root with a View

The Associate Root With View **(GPARV)** subroutine lets your application associate a structure network, with a view and, implicitly, a workstation. It also lets your application specify the priority of the root structures within that view.

Similar to the priority of views, the graPHIGS API processes root structures within each view in reverse priority order. This results in higher priority networks overlaying lower priority networks on the display surface. This gives your application the following three ways of controlling display priority:

1. The priority of views within a workstation
2. The priority of structure networks within a view
3. The sequence of primitives within a structure network

Disassociating Structure Networks from Views and Workstations

The graPHIGS API provides program subroutine calls that let your application alter the relationships between structure networks, views, and workstations. None of the subroutine calls in this section delete a root structure network from the centralized structure storage. They only modify the relationship between the root and the specified workstation.

Disassociate Root from Workstation

The Disassociate Root from Workstation **(GPDRW)** subroutine lets your application remove a specified structure from a specified workstation. The **GPDRW** subroutine is the logical reverse of the Associate Root With Workstation **(GPARW)** subroutine. If the structure is no longer referenced by any other structure on this workstation, the API removes that structure from the workstation. It also removes all of its sub-structures if those sub-structures do not belong to another root's network. The **GPDRW** also removes that structure from all views.

Disassociate All Roots from Workstation

The Disassociate All Roots from Workstation **(GPDARW)** subroutine lets your application remove all structures from a specified workstation. Upon completion of the **GPDARW** subroutine, the workstation's list of associated roots is empty. Thus, it also empties all views.

Disassociate Root from View

The Disassociate Root from View **(GPDRV)** subroutine lets your application remove a specified root from a view on a specified workstation. The root remains associated with the workstation. The **GPDRV** is the logical reverse of the associate root with view **(GPARV)** subroutine.

Disassociate Root from All Views

The Disassociate Root from All Views **(GPDRAV)** subroutine lets your application remove a specified root from all views on a given workstation.

Empty View

The Empty View **(GPEV)** subroutine lets your application remove all the roots from a specified view. Upon completion of the **GPEV** subroutine, the specified view exists but displays no graphics data.

Empty All Views

The Empty All Views **(GPEAV)** subroutine lets your application remove all the roots from all the views at a specified workstation.

Structure Traversal

After associating structure networks with views and workstations, your application can initiate the process of displaying an image on an output device. This process involves a sequential traversal of the structure elements.

In the sample program introduced in Chapter 1, after associating the structure to the workstation and view, the statement:

```
CALL GPUPWS(WSID,2)
```

initiates the structure traversal process and generates the house drawing on the workstation display.

Structure Traversal Processing

As shown in the following figure, each view is traversed in the order of its assigned priority, from lowest to highest. Within each view, root structures and their corresponding networks are likewise processed in reverse priority order. Therefore, structure traversal begins by processing the lowest priority root structure in the lowest priority view.

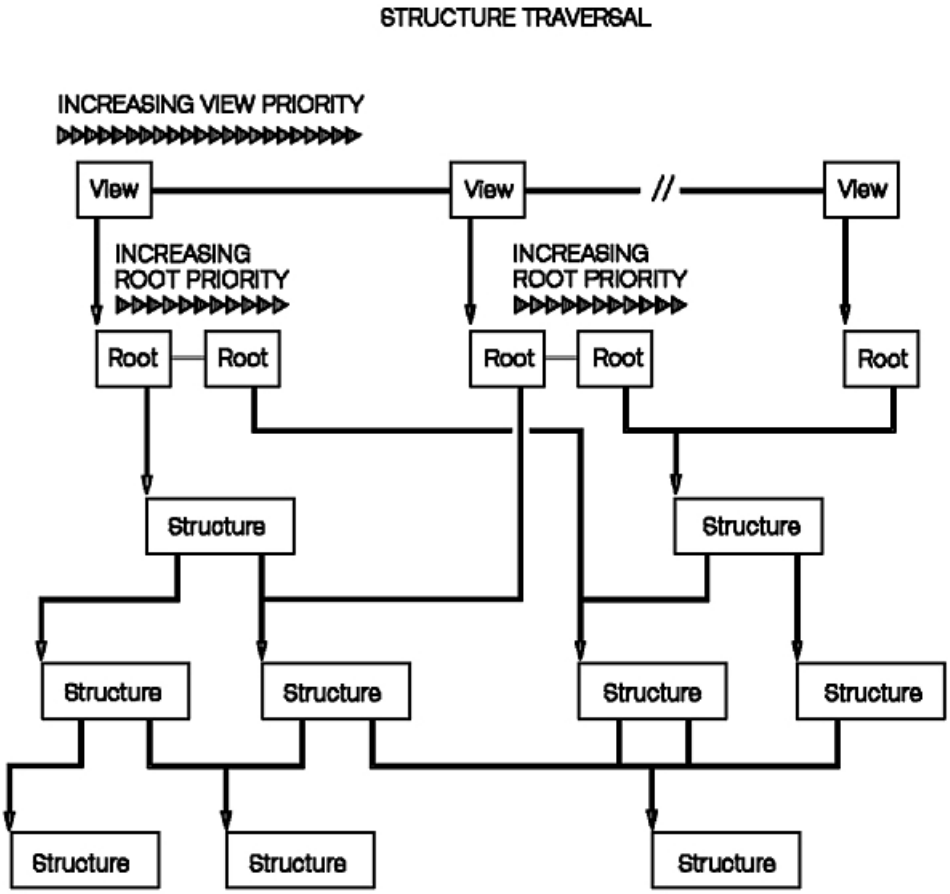


Figure 27. Structure Traversal Processing. This illustration shows how each view is traversed in the order of its assigned priority, from lowest to highest.

Each time the traversal of a root's structure network is initiated, a default set of attribute and transformation values exists in a conceptual set of *traversal-time registers.* For example, the default composite modeling transformation matrix is the identity matrix, and the default line type is `1=SOLID_LINE`. The complete set of traversal-time default values can be found in *The graPHIGS Programming Interface: Technical Reference*.

Structure elements which set attributes or define modeling transformations may modify these conceptual registers. Other structure elements of the same structure, or of another structure invoked by this same structure, may use the values in these conceptual registers.

The traversal of a root's structure network begins at element one of the root structure. Elements are then processed sequentially, ending with the last element of the root structure. After completing the processing of a root structure, the API proceeds to traverse either the root with the next highest priority, or if this was the last root in the current view, it continues to the next highest priority view. The traversal process ends when the highest priority view is completely traversed.

As the graPHIGS API processes the structure networks, it encounters elements sequentially. As it encounters these elements, the effect of processing a given element results in one of the following four actions:

No Action
The label and application data structure elements have no traversal-time effect.

Modification of Traversal-Time Registers
Structure elements that modify traversal-time registers include attribute setting elements and modeling transformation elements.

Drawing a Graphics Primitive
Output primitive structure elements, when encountered cause the API to draw the primitive using the current attribute and transformation settings.

Invocation of Another Structure
The structure element, **GPEXST**, causes the API to traverse the identified structure at that point.

Each time the graPHIGS API encounters an output primitive, it applies the current attribute and transformation values to that primitive. The current values reflect changes that are due to structure element assignments or inheritance. This is called *traversal-time attribute binding*

Barring any local attribute assignments, when a primitive is referenced by multiple structures, the appearance of that primitive is determined, in each instance, by the inherited attributes.

When encountering a **GPEXST** element within a structure, the system suspends traversal of that structure. It also saves the current transformation and attribute values. Then it completely traverses the invoked structure. On return to the structure, the API restores the state of the transformation and attribute values, then resumes structure traversal.

Resolving Attribute Specification During Structure Traversal

During structure traversal, the system keeps track of the current attribute values in the traversal-time registers. When encountering an output primitive, the system checks the ASF setting of each attribute to determine whether it obtains that attribute from the system's conceptual attribute registers, or from a workstation's bundle table. These conceptual registers contain either an attribute specification or an index into a bundle table.

The current values can consist of default values, inherited values, and assigned values as a result of any attribute assignment element.

The graPHIGS API Transformation Pipeline and Coordinate Systems

Each time the system encounters an output primitive, it passes that primitive's coordinate data through a series of transformations before displaying output on a device. This series of transformations is called the **transformation pipeline** depicted in the following figure:

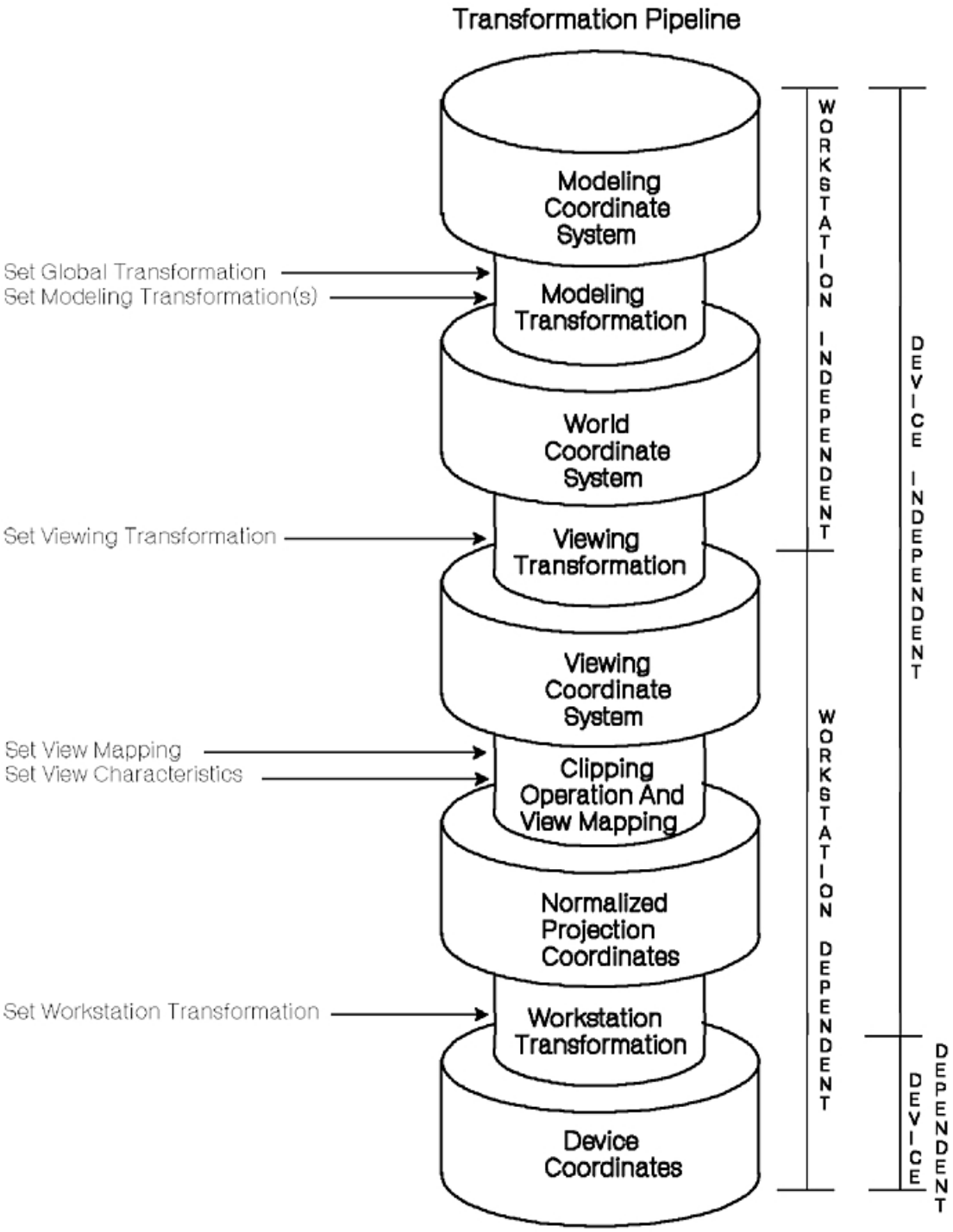


Figure 28. Transformation Pipeline With Relation to the Various Coordinate Systems. This illustration shows the various stages that the primitive's coordinate data must pass through before being displayed as output on a device. The Modeling Coordinate System must go through Modeling Transformation. The coordinates will now be in the World Coordinate System. These coordinates go through a Viewing transformation to get the coordinates in the Viewing Coordinate System. The coordinates then go through clipping operation and view mapping to get into Normalized Projection Coordinates. Finally, these coordinates go through Workstation Transformation to be in Device Coordinates. The stages beginning with the Modeling Coordinate System to the Viewing Transformation are all Workstation Independent while the stages beginning with the Viewing Coordinate System and ending with Device Coordinates are Workstation Dependent. The various stages between the Modeling Coordinate System to the Workstation Transformation are Device Independent while Workstation Transformation and Device Coordinates are Device Dependent. The values used to compute the transformations can be set. Modeling Transformation can be altered by setting the Global Transformation and the Modeling Transformation(s). The Viewing Transformation can be altered by setting the Viewing Transformation. The Clipping Operation and View Mapping can be altered by setting the View Mapping and View Characteristics. The Workstation Transformation can be altered by setting the Workstation Transformation.

The treatment of this subject is done here at a conceptual level. How the implementation actually realizes the capabilities described here may be significantly different from the conceptual, stepwise process described here. For example, an intelligent workstation may be able to collapse consecutive transformations into only one transformation. Most important to the application programmer is that the system always functions identical to this conceptual description.

Data does not pass through the pipeline until it is time to display the data.

The transformation pipeline begins with the coordinate system within which your application defines its graphics data, the Modeling Coordinate System (MCS). The pipeline ends with the Device Coordinate System (DCS), which provides the coordinates used by the output workstation hardware.

The pipeline consists of five distinct coordinate systems with four transformation processes which transform that data from the MCS to the DCS.

In the trivial case, a step in the pipeline is equal to an identity transformation. An identity transformation is the system's default for both modeling and viewing transformations. In this case, the transform does not affect the data in any way.

Only the modeling transformations can be modified by structure elements during structure traversal. All other transformations are set by way of control subroutines that are contained in tables. With the exception of the modeling transformation value, all transformation values are established when structure display traversal is initiated. The following sections discuss the function of each step in the transformation pipeline.

Modeling Coordinates and Transformations

The graPHIGS API defines all primitives in the Modeling Coordinate System (MCS). Your application may use any portion of modeling space. Modeling space is only bounded by the range representable by a 32-bit real number.

A typical usage is to define primitives in different portions of modeling space and then "assemble" them to create the final model. The "assembling" of parts of the model is done using modeling transformations. They transform the primitive's coordinates into a common coordinate system called the World Coordinate System (WCS).

Modeling transformations modify the coordinates of a primitive in an application-defined manner. The modeling transformation (like all transformations) is represented by a *4 [default] 4 matrix.* Modeling transformations may translate the data, scale the data, rotate the data, or any other operation possible by the multiplication of a coordinate by a 4 [default] 4 matrix. The default modeling transformation is an identity transformation.

World Coordinates and the Viewing Transformation

After the modeling transformation is applied, all data is in the World Coordinate System (WCS). Like modeling space, world coordinate space is only bounded by the range representable by single precision floating-point numbers.

When your application defines a view it indicates to the system what portion of the World Coordinate System it wants to display.

By means of the **viewing transformation**, world coordinate data is transformed into the Viewing Coordinate System. Your application can use the viewing transform to rotate, scale, or in some way modify the world coordinate data and transform it into the Viewing Coordinate System.

Viewing Coordinates and the Clipping and View Mapping Operations

Any combination of structures may be viewed using the same view definition. Your application can use a structure in many views. More than one view may exist for each workstation and their definitions are contained in the view table. There is one view table for each workstation.

It is from the viewing coordinate system that the projection operation takes place. The type of projection (or view) of the data may be parallel or perspective. The region to be projected is defined by two clip planes, a window, a projection reference point (eye point), a view plane position (for projecting onto) and a projection type.

After the data is projected it is mapped, by way of the defined view mapping, into the Normalized Projection Coordinate (NPC) system. The view mapping consists of a window on the view plane near/far clip planes and a viewport in NPC space.

NPC Coordinates and the Workstation Transformation

As the name implies, NPC space is a normalized coordinate system. NPC extends from 0.0 to 1.0 in the X, Y and Z directions. A viewport, in this space, is defined as minimum and maximum X, Y and Z values which define a viewport volume in NPC space.

The final step in the transformation of the data to the display takes the data from NPC space to the coordinate system of the device. The workstation transformation takes the data from NPC to Device Coordinates (DC). There is one workstation transformation per workstation. The workstation transform positions the logical workstation's display surface on the physical display. In the default case, all of NPC is mapped onto all of DC. Most applications do not modify this relationship.

Device Coordinates

Device Coordinates (DC) are typically defined in meters and express the actual size of the physical display. Another means of describing the display characteristics is address units (rasters) which measure the actual addressability of the display. Both the address units and device coordinates used by a workstation may be determined by inquiries.

Direct Window Mapping

There are two methods the graPHIGS API uses to display data in an X-Window: `1=MAPPED` and `2=DIRECT` Use the Set Device Coordinate Mapping Method **(GPDCMM)** subroutine to choose the method that best suits your application.

By default, the graPHIGS API uses the `1=MAPPED` display method. This method displays data in an X-Window by scaling the graPHIGS API device coordinate (DC) range to fit the X-Window. When you change the X-Window size, the graPHIGS API scales the contents to fit the new window. In this way the aspect ratio of the data is preserved, the same amount of data is displayed regardless of scale, and the window can be said to behave like a ″rubber sheet.″

When using the DIRECT method, the graPHIGS API displays the DC range directly in the X-Window with no scaling. The lower-left corner of the DC volume is aligned with the lower-left corner of the window. If the window is smaller than the DC range addressed in the data, then the window clips the data. If the window is larger than the display data, the area of the window beyond the DC range of the data is unused. This window behavior can be likened to that of a ″porthole.″

Refer to *The graPHIGS Programming Interface: Technical Reference* for more information on the `2=DIRECT` method of display in an X-Window.

Resolving Modeling Transformations During Structure Traversal

During structure traversal, the system keeps track of the current global and local modeling transformation matrixes in conceptual registers. When root structure traversal begins, the default matrix for both the global and local matrixes is the identity matrix.

During structure traversal, each substructure inherits only one composite transformation matrix. The composite matrix is the concatenation of the local modeling transformation and the global transformation at the time the **GPEXST** element is encountered. The graPHIGS API refers to the composite matrix in the

parent structure as the global modeling transformation matrix of the invoked structure. The following figure depicts how structures beginning with the tabletop structure inherit transformation matrixes:

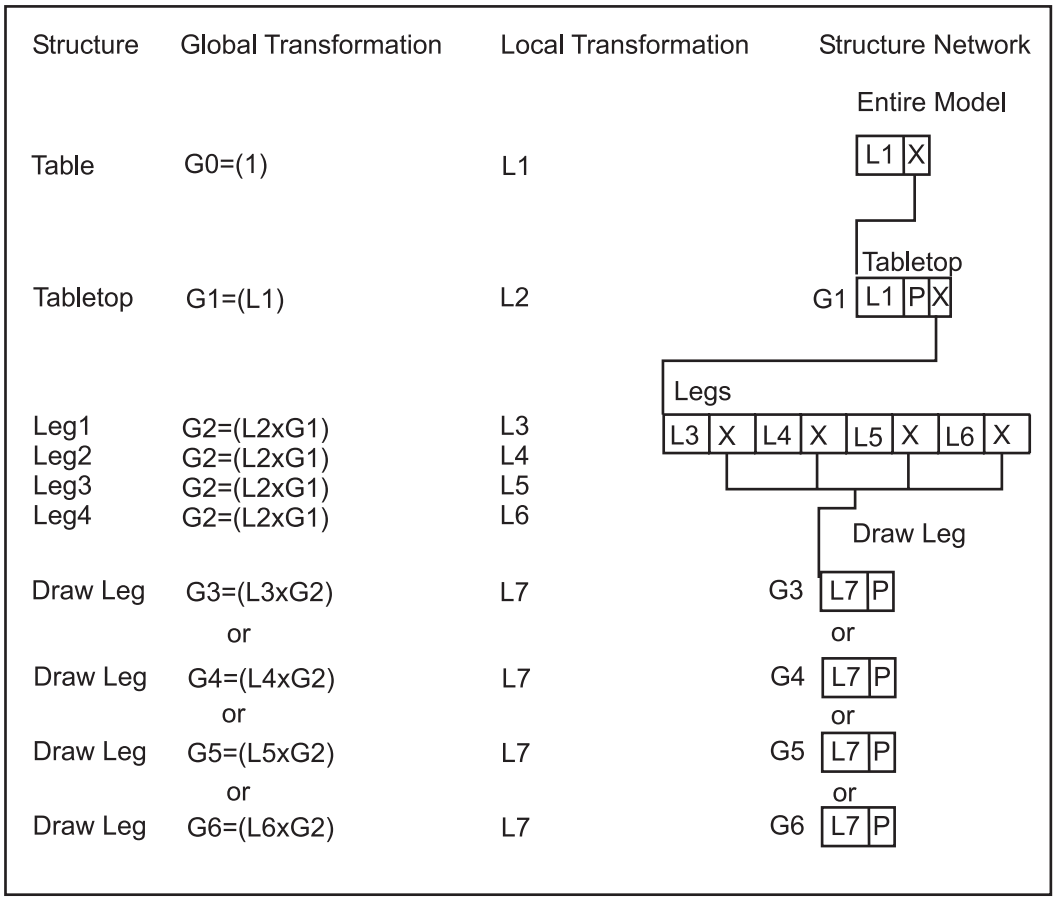


Figure 29. How Structures Inherit Transformation Matrixes. This illustration shows how structures beginning with the tabletop structure and ending with drawing the legs inherit transformation matrixes. The table structure used in this illustration is the same one described in the figure from Chapter 3 entitled Hierarchical Structure Relationships. The global transformation used for the table structure is G0=(1) and the local transformation is L1. The transformation n L1 is applied and the tabletop structure is executed. The tabletop structure uses global transformation G1=(L1) and local transformation L2. The G1 transformation is applied before the L1 transformation for the tabletop, then the L1 transformation is applied to the primitive(P) and the legs structure is invoked. The global transformation for the legs structure is G2=(L2xG1). The local transformation for Leg1 structure is L3, L4 for Leg2, L5 for Leg3, and L6 for Leg6. The Legs structure is executed in the following order: local transformation L3 is applied, the Draw Leg structure is invoked, L4 is applied, the Draw Leg structure is invoked, L5 is applied, the Draw Leg structure is invoked, L6 is applied, and the Draw Leg structure is invoked one last time. The first time that Draw Leg is executed, the global transformation is defined as G3=(L3xG2), the second time G4=(L4xG2), the third time G5=(L5xG2), and the fourth time G6=(L6xG2). The local transformation for each draw leg structure is defined as L7. For the execution of the first leg structure, G3 is applied, L7 is applied, and finally the leg primitive(P) is drawn. For the second leg, G4 is applied, L7 is applied, and then the second leg is drawn. For the third, G5 is applied, L7 is applied, and then the third leg is drawn. For the fourth leg, G6 is applied, L7 is applied, and then the fourth leg is drawn.

- ″I″ means identity matrix
- ″G″ means global transformation
- ″X″ means invoke another structure **(GPEXST)**
- ″L″ means concatenate local transformations prior to invoking another structure
- ″P″ means primitive.

When encountering the tabletop primitive, the system multiplies the tabletop's Modeling Coordinates (MC) by the concatenation of L2 and G1.

When encountering the leg primitive, the system multiplies the leg's coordinates by the concatenation of the appropriate inherited matrix and the leg's local modeling transformations. Notice that L3 through L6 all use the replace concatenation type option. As discussed in Chapter 3. Structures, the concatenation type option specifies how the modeling transformation element operates on the current modeling transformation values.

Drawing the first leg entails multiplying the leg's coordinates by a concatenation of L7 and G3.

Drawing the second leg entails multiplying the leg's coordinates by a concatenation of L7 and G4.

Drawing the third leg entails multiplying the leg's coordinates by a concatenation of L7 and G5.

Drawing the fourth leg entails multiplying the leg's coordinates by a concatenation of L7 and G6.

Your application may override the inherited global transformation by using the Set Global Transformation (**GPGLX2** and **GPGLX3**) subroutine. The new global transformation matrix replaces the previous global matrix.

Resolving Detectability, Highlighting, and Invisibility

Primitives may be grouped by your application into application-defined classes. By using filters, provided by the graPHIGS API, your application may control the detectability, highlighting and invisibility of these primitives. During structure traversal, the system manages the current class name values and resolves which primitives are detectable, highlighted, or visible, based on the primitive's class specification.

To this end, the system supports the following three inclusion and exclusion filters:

- pick filter (one pair per pick device)
- highlighting filter (one pair per workstation)
- invisibility filter (one pair per workstation)

These filters operate independently of each other. Making a particular class of primitives detectable, for example, does not affect whether that class is highlighted or visible.

To provide dynamic control over which classes are detectable, highlighted, and visible, each filter contains an inclusion filter and an exclusion filter.

These filters control the detectability, highlighting, and invisibility of entire classes of output primitives.

The system provides a consistent set of rules that govern the interrelationships between classes assigned to inclusion and exclusion filters. An output primitive is detectable, highlighted, or invisible if and only if both of the following are true:

- It is a member of a class that is specified in a corresponding inclusion filter.
- None of its classes are contained in a corresponding exclusion filter.

When all of the filters are empty, for example, at initialization time:

- primitives are not detectable.
- primitives are not highlighted.
- primitives are not invisible (they are visible).

Inclusion and exclusion filter designations are control subroutine calls that modify the workstation state. The following figure depicts a root structure with class designations:

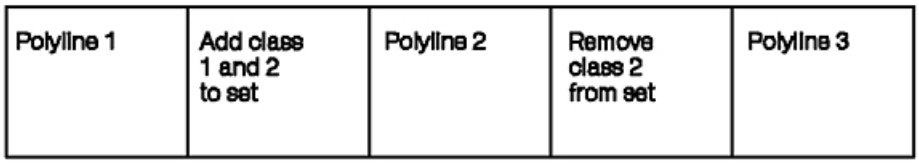


Figure 30. Root structure with class designations. This illustration shows the root structure containing polyline 1, add class 1 and class 2 to set, polyline 2, remove class 2 from set, and polyline 3.

By proper organization of structure content, your application can modify these characteristics of primitives without structure editing. In the figure, Root structure with class designations, polyline 1 can not be made detectable, highlighted, or invisible. Polyline 2 is affected by how your application assigns classes 1 and 2 to inclusion and exclusion filters. Polyline 3 is affected only by how your application assigns class 1 to inclusion and exclusion filters.

Pick Filter: The Set Pick Filter **(GPPKF)** subroutine enables your application to specify which classes belong to a pick device's inclusion and exclusion filters. Using the information in the figure, Root structure with class designations, an assignment of class 1 to only the pick inclusion filter, makes both polyline 2 and polyline 3 detectable.

Highlighting Filter: The graPHIGS API enables your application to control the highlighting of primitives. The Set Highlighting Filter **(GPHLF)** subroutine enables your application to specify which classes belong to highlighting inclusion and exclusion filters. Using the information in the figure, Root structure with class designations, if your application only assigns class 2 to the highlighting inclusion filter, then only polyline 2 is highlighted. Because polyline 3 only belongs to class 1, it is affected only by those changes that affect class 1 because class 2 is removed before polyline 3 is drawn.

Invisibility Filter: The graPHIGS API lets your application specify whether a primitive or set of primitives are visible. The Set Invisibility Filter **(GPIVF)** subroutine lets your application control which classes are in a workstation's inclusion and exclusion filters. Remember that the default is the absence of classes in either filter. This means that all primitives are initially visible. Using the information in the figure, Root structure with class designations, if your application assigns class 1 to the invisibility inclusion filter and class 2 to the invisibility exclusion filter, then polyline 3 becomes invisible. In order to analyze why polyline 2 remains visible remember the rules that govern how filters affect primitives. Polyline 2 remains visible because at least one class (number 2) is in the invisibility exclusion filter. This excludes polyline 2 from becoming invisible.

Updating the Workstation

The graPHIGS API provides two methods for your application to assure that the display image corresponds exactly to the state of the system.

The Redraw All Structures **(GPRAST)** subroutine forces a new frame action on the specified workstation, and then rebuilds the display based on the current state of the system. A new frame action involves clearing the screen on the display or feeding a new sheet of paper on a plotter-type device.

The Update Workstation **(GPUPWS)** subroutine assures that the display exactly matches the state of the system. A new frame action occurs only if necessary.

When **GPUPWS** is called, all deferred actions for the specified workstation are executed without an intermediate clearing of the display surface. If the regeneration flag is set to PERFORM and a new frame action is necessary the following actions are taken on the specified workstation:

1. If the display surface is not empty the display is cleared.

2. If the workstation window, viewport, view matrix, view mapping, or view characteristics have changed for any view, the current entries in the Workstation State List (WSL) are assigned the corresponding values from the requested WSL entries.
3. All structures associated with views on the workstation are re-displayed.

With the Update Workstation Asynchronous **(GPUPWA)** subroutine, transactions with the specified workstation do not wait for an on-going traversal to complete. This means that a traversal may be interrupted before completion, causing intermediate screen changes to go unseen. (It may be important, for example in animation sequences, to see all updates to the workstation in a sequential fashion. In this case, the synchronous Update Workstation **(GPUPWS)** subroutine should be used. Use **GPUPWA** when it is necessary to see the most up-to-date screen as soon as possible).

When there are no changes pending at the workstation, and GPUPWS is called, no action occurs. On the other hand, if your application invoked **GPRAST** in this situation, the workstation clears the display surface and re-displays the same image. On a plotter-type device, this can be used to generate many identical plots.

Deferral Modes

Deferral modes let your application control when the graPHIGS API traverses the structure networks and generates an image on a particular workstation. The following deferral modes let your application control the deferral state of the system:

1=ASAP
As soon as possible

2=BNIG
Before the next interaction globally

3=BNIL
Before the next interaction locally

4=ASTI
At some time

5=WAIT
When the application requests it (default).

These modes do not affect when structure editing occurs, rather they affect when structure modifications are seen on the display surface. Since your application can edit structure networks without changing the output image, the image may not reflect the current graphics data within the structure networks. These deferral modes let your application control when to resolve the differences between the structure definitions and the image.

When the deferral mode is satisfied, the API traverses the structures and generates an image that corresponds to the graphics data in the structure networks and the current settings in the workstation's tables, that is, a **GPUPWS** is performed.

Your application may choose to defer changes to an image when a lot of editing occurs, in order to buffer the changes and optimize the transfer of data.

Remember that the **GPUPWS** subroutine causes an immediate traversal of the structure networks and generation of an image on an output device, no matter what deferral mode is in effect.

Deferral modes have many application-defined uses. For example, your application may choose to prevent an image change until a particular condition is satisfied.

1=ASAP Deferral Mode: When a workstation is in ASAP deferral state, the API re-traverses all data after every program subroutine call and updates the output image. This state introduces the maximum system overhead per subroutine call.

2=BNIG Deferral Mode: A workstation in BNIG mode is updated immediately before an interaction occurs at any open workstation. For example, suppose workstation one is in ASAP mode and workstation two is in BNIG mode. If an output display operator interacts with workstation one, the system updates workstation two before the interaction takes place.

Notice that when your application uses BNIG and it places an input device in Sample or Event mode, the system always re-traverses the structure networks, since on every subroutine call an interaction is always underway. Until the input device mode is changed, the BNIG mode simulates the ASAP mode. This may adversely affect performance.

3=BNIL Deferral Mode: A workstation in BNIL mode is updated immediately before an interaction occurs at that workstation.

For example, suppose that workstation one is in ASAP mode, workstation two is in BNIG mode, and workstation three is in BNIL mode. If an output display operator interacts with workstation two, then the system updates workstation one and two, but not three. However, if the display operator interacts in workstation three, then the API updates all three workstations.

Notice that when your application uses BNIL and it places an input device in sample or event mode, the system re-traverses the structure networks, since on every subroutine an interaction is always underway. Until the input device mode is changed, the BNIL state simulates the ASAP mode. This may adversely affect performance.

4=ASTI Deferral Mode: Changes to the image of a workstation in ASTI mode occur in an implementation-dependent way. This implementation treats ASTI the same as BNIL**.**

5=WAIT Deferral Mode: The WAIT deferral mode is the default mode. In this mode, all changes to a workstation are deferred until your application explicitly requests them. Your application can cause changes to a workstation in WAIT state by:

- changing the deferral mode to one that necessitates a re-traversal such as ASAP
- issuing a **GPUPWS** subroutine

Notice that, in this mode, your application defers all changes to a workstation until the **GPUPWS** subroutine is issued.

Modification Modes

The graPHIGS API supports three Modification Modes selectable through the Set Deferral State **(GPDF)** subroutine.

1=NO_IMMEDIATE_VISUAL_EFFECT (NIVE)
Establishes an environment in which changes to the image depend on the deferral mode only.

2=UPDATE_WITHOUT_REGENERATION
Allows modifications to the display that require no retraversal.

3=QUICK_UPDATE
Useful for trivial updates to large models. (See discussion below.)

For information on the use of these modes, see the *The graPHIGS Programming Interface: Writing Applications*.

QUICK_UPDATE

QUICK_UPDATE mode allows the graPHIGS API, running with XLIB and XDWA device drivers, to show the results of structure editing functions by direct screen modification, eliminating the need for a complete screen redraw. Use of QUICK_UPDATE should be weighed against the need for thoroughly accurate screen contents, since the results of a QUICK_UPDATE may not be perfectly displayed.

When operating in QUICK-UPDATE mode, the application performs as many structure editing operations as possible without redrawing the screen. The success of a QUICK_UPDATE depends on the target structure element and the editing operation. Class name, for example, cannot be changed using QUICK_UPDATE. When the graPHIGS API finds that an operation leaves the display too inaccurate, it terminates the QUICK_UPDATE operation and redraws the screen.

Refer to *The graPHIGS Programming Interface: Technical Reference* for more information on QUICK_UPDATE.

Chapter 7. Input Devices

Up to now you have seen how to display your graphic model, you will now learn how to make your application interactive. To handle interactive input, the graPHIGS API supports various input devices, giving your application a great deal of flexibility with regard to how your application interacts with an input device operator.

Advanced topics related to input processing are discussed in Chapter 18. Advanced Input and Event Handling in Part Two. This section describes the basic input device concepts provided by the graPHIGS API.

Modes of Interaction

The graPHIGS API supports three modes (Operating Modes) of interaction:

- 1=REQUEST
- 2=SAMPLE
- 3=EVENT

REQUEST Mode Input

This mode establishes a synchronous environment between your application and a chosen device. It is the default mode for all devices. In this mode, your application must explicitly request input from a particular device. In REQUEST mode, the device is inactive until your application explicitly requests input from that device.

After the request is made the device is activated and your application waits until the operator either enters the requested input or performs a break action. Pressing a reset-type key is an example of a break action. The break action depends on the logical input device and on the particular workstation. If a break occurs, the logical input value is not valid.

There can be only one outstanding request at a time. A value returns when the operator triggers the input device.

In the sample program introduced in Chapter 1, you have used the REQUEST mode of interaction. The statement:

```
CALL GPRQCH(WSID,1,STATUS,CHOICE)
```

causes the program execution to halt and wait for the operator to trigger a choice device. Your program will terminate if the variable Status contains a 1, that is, if the operator performs a break action.

SAMPLE Mode Input

When your application puts a device in SAMPLE mode, it is immediately activated for manipulation by a workstation operator. Your application obtains the current values of the input device by explicitly sampling it. For example, your application can obtain the location of the screen cursor at any point in time, through the use of SAMPLE mode input.

EVENT Mode Input

This mode establishes an *asynchronous* environment between your application and a chosen device. In this mode, both your application and any corresponding device operate independently of each other. An input queue exists which accumulates events from all devices that are in EVENT mode, on all workstations. Any device in EVENT mode can supply data to the input queue. Your application can retrieve input from the queue without affecting input device operation. Notice that the input queue is centralized and collects events from all open workstations.

The graPHIGS API maintains one queue containing the input of all devices. The input is queued, first in, first out, in an *event report* form. The event report contains the input data and an identification of which device sent the input.

An event is generated each time an input device is triggered by an operator. For example, the trigger of a locator device might be the tip switch on a stylus. Notice that one trigger may cause event reports from multiple devices to be queued. This means that a single operator action can cause multiple events. These are called *simultaneous events* For example, the pick device and locator device may share the same trigger. If both devices are in `EVENT` mode and their common trigger is fired, it generates simultaneous `PICK` and `LOCATOR` events, which are placed on the event queue.

Before your application can get queued input, it must first transfer the next item on the queue to a current event report area. The Await Event **(GPAWEV)** subroutine transfers the next item on the queue, if one exists, to the current event report area.

If no events are queued, the API waits for a time interval specified by your application in the **GPAWEV** subroutine. Control returns to your application at the end of the time interval, or when an event arrives on the queue. After issuing a **GPAWEV** subroutine, your application can examine the input data contained in the current event report. This gives your application the opportunity to either process the input data or discard it by issuing another GPAWEV subroutine.

To retrieve the data, your application issues a **get data** programming subroutine call. There is a different **Get** subroutine call for each type of queued input data. Getting data is discussed in each “Device Operation” section.

Your application can flush the queue of all event reports that belong to a specified device on a specific workstation using the Flush Device Events **(GPFLEV)** subroutine.

Your application can also flush all event reports for all devices on a specified workstation using the Flush Workstation Events **(GPFWEV)** subroutine.

As discussed in Part Two of this book, the event queue is also used in the processing of asynchronous events other than those generated by input devices.

Modified Sample Program 1

Modify your sample program according to the following instructions:

1. In the DECLARE VARIABLES section, please add the following statements:

```
REALx4 AREA(6),CSIZE(3)
INTEGERx4 DATAL,DATA(4),ERRIND,UNITS,ASIZE(3)
CHARACTERx8 ACONID, AWSTYP
INTEGERx4 IWSID,CLASS,DEVICE,ILEN,OLEN
DATA DATAL /16/
DATA DATA /1,0,0,2/
DATA ILEN /8/
```

2. Replace all the INPUT SUBROUTINES statements with the following:

```
1     CALL GPQRCT(WSID, ILEN , ERRIND, OLEN, ACONID, AWSTYP)
      CALL GPQDS(AWSTYP, ERRIND, UNITS, CSIZE, ASIZE)
      IF(ERRIND.NE.0) GOTO 200
      AREA(1) = 0.
      AREA(2) = CSIZE(1)
      AREA(3) = 0.
      AREA(4) = CSIZE(2)
      AREA(5) = 0.
      AREA(6) = CSIZE(3)
      CALL GPINCH(WSID,1,1,2,AREA,DATAL,DATA)
      CALL GPCHMO(WSID,1,3,2)
      TIME = 10.
100   CALL GPAWEV(TIME,IWSID,CLASS,DEVICE)
      IF(CLASS.EQ.4) THEN
        IF(DEVICE.EQ.1) THEN
          CALL GPGTCH(CHOICE)
          IF(CHOICE.EQ.1) GOTO 200
        ENDIF
      ENDIF
      GOTO 100
```

Once you have done the modifications, compile and run your program.

You will notice that the first button of the Lighted Program Function Keyboard (LPFK) is lighted. Press it. The program should terminate. The break action procedure is of no use now, your program will only terminate if the LPFK button is pressed.

The modifications you added initialized the LPFK CHOICE device by using the **GPINCH** subroutine. Then, the **GPCHMO** defined the EVENT mode as the interaction mode for this device.

The **GPAWEV** subroutine allows your application to examine the event queue. Pressing the LPFK button generates an event of class 4. Your application then checks to determine if the first button (CHOICE.EQ.1) was pressed. If so, the program terminates. If not, the event queue is examined again.

Device Classes

The graPHIGS API supports six classes of input devices. Each device class represents a generic physical input device.

- 1=CHOICE
- 2=LOCATOR
- 3=PICK
- 4=STROKE
- 5=STRING

- 6=VALUATOR

Because these device classes represent generic physical devices, the API must provide a way for your application to determine the actual capabilities of a specific physical device. Your application can use the appropriate inquiry programming subroutine calls to determine these specific capabilities. Which devices are supported, their supported prompt echo types, and the maximum workstation supported buffer sizes are all examples of input device characteristics that can be determined using the appropriate inquiry subroutine calls.

These capabilities are summarized for each workstation type in *The graPHIGS Programming Interface: Technical Reference*. The triggers used by each input device for each workstation type can also be found there. Nevertheless, keep in mind that those capabilities may vary, depending on the workstation's configuration. To be sure of a workstation's actual capabilities, use the inquiry subroutines.

CHOICE Device

This section discusses how your application manages a CHOICE device in the graPHIGS API environment. A CHOICE device provides a positive integer that represents a selection from a number of choices. Zero indicates no choice.

The Lighted Program Function Keyboard (LPFK) on the IBM 5080 is an example of a CHOICE input device. The CHOICE prompts on this device are the keys' lights. A lit key typically indicates that the application program uses this key. The CHOICE device is triggered when an operator presses any key.

Establishing the Operating Mode of a CHOICE Device

Your application selects an operating mode using the Set Choice Mode **(GPCHMO)** subroutine. **GPCHMO** lets your application specify whether the CHOICE device is in REQUEST, SAMPLE, or EVENT mode. Your application defines the operating mode of each CHOICE device, independently. For example, your application might put one CHOICE device, such as a lighted program keyboard in REQUEST mode, and a second CHOICE device, such as the buttons on a cursor controller in EVENT mode. The default mode is REQUEST.

Providing Initial Values for a CHOICE Device

Your application initializes the device through the Initialize Choice **(GPINCH)** subroutine. For example, **GPINCH** lets your application specify which LPFKs to light. Your application can use this subroutine only when the device is in REQUEST mode, that is, when the device is not active.

In the sample program, the statement:

```
CALL GPINCH(WSID,1,1,2,AREA,DATAL,DATA)
```

initialized the CHOICE device. The second parameter in this statement indicates to the system that we want to initialize the Lighted Program Function Keyboard (LPFK). The last two parameters indicate which buttons we want to use, thus, the DATA /1,0,0,2/ indicates that we want to light (2=ON) the first button of the 32 available in the LPFK.

Some CHOICE devices provide echo types which utilize portions of the display screen. For example, the current CHOICE number can be displayed on the screen. The region of the display surface used for this echo is specified by the echo area parameter of the **GPINCH** subroutine. The input device echo is clipped to the application specified echo area.

In the **GPINCH** statement of the sample program, the parameter AREA indicates the part of the screen we want to use as the echo area, using Device Coordinates (DCs). The LPFK does not produce any echo on the screen; nevertheless, the parameter must be present.

In the sample program you are making use of the inquiry facility of the graPHIGS API The statements:

```
CALL GPQRCT(WSID,ILEN,ERRIND,OLEN,ACONID,AWSTYP)
      CALL GPQDS(AWSTYP,ERRIND,UNITS,CSIZE,ASIZE)
```

allow your application to inquire the size, in Device Coordinates (DCs), of the display screen available on your workstation. The size is returned in the parameter CSIZE Your application then uses this information to initialize the ***variable Area*** needed in the **GPINCH** subroutine. This facilitates the programming of device independent applications.

Input device initialization subroutine calls may be called only when the device is in REQUEST mode, that is, when the device is not activated.

Obtaining Input from a CHOICE Device

The program subroutine call needed to obtain input from a CHOICE device depends on the Operating Mode of that device.

Obtaining Input from a CHOICE Device in REQUEST Mode: In REQUEST mode, your application is responsible for initiating all interactions with the CHOICE device operator. If your application needs choice input from an operator before it can proceed, it asks the operator for input by issuing a Request Choice **(GPRQCH)** subroutine call.

After issuing the request, the graPHIGS API waits for the operator to satisfy the request. The returned information indicates the status of the input and number of the button that was pushed.

After the requested information is obtained, the system returns control to your application so it can resume processing and determine how to use the returned information.

Obtaining Input from a CHOICE Device in SAMPLE Mode: Setting a CHOICE device in SAMPLE mode activates the CHOICE device. When your application needs the current value, it issues a Sample Choice **(GPSMCH)** subroutine. When there is no input, the system continues to report that there is no input.

When your application issues a **GPSMCH** subroutine, it receives the current CHOICE value as fast as the system can supply it. However, lacking synchronization, the use of a CHOICE device in this mode has limited utility.

Obtaining Input from a CHOICE Device in EVENT Mode: Setting a CHOICE device in EVENT mode activates the device and establishes an indirect relationship between your application and a CHOICE device that is mediated by the event queue.

An operator triggers CHOICE events which are placed on the event queue. Your application can then get that input from the event queue. This lets operator input and application processing occur asynchronously.

Upon issuing the **GPAWEV** subroutine, the API gives your application the identification of the device corresponding to the current event report. If the input came from a CHOICE device and your application wants to use that input, your application issues a Get Choice **(GPGTCH)** subroutine. GPGTCH returns the value of the CHOICE input.

In the sample program, the statement:

```
CALL GPGTCH(CHOICE)
```

gets input from the current event report. The CHOICE parameter then has a numeric value which identifies the CHOICE device key pressed by the operator. Your application can then proceed to use this value within the program logic.

LOCATOR Device

A LOCATOR device provides a position in World Coordinates (WC). Also returned with the LOCATOR input is identification of the view whose matrix was used to convert the screen position indicated back to World Coordinates. For a given screen location, the active view of highest priority is used for this conversion. Generally, all LOCATOR input occurs from View 0, the default highest priority view, unless this relationship is

changed using the Set View Input Priority **(GPVIP)** or Set View Priority **(GPVP)** subroutine. The tablet and cursor controller are an example of a LOCATOR input mechanism.

Typically, a LOCATOR is a 2D device from which a 3D coordinate is derived. This is done as follows:

The LOCATOR device conceptually fires a ray along the direction of projection into the display surface. The intersection of this ray with the front of the effective clipping volume is the location returned. The front of the clipping volume is typically the near clip plane. By modifying the workstation transform, an application may position the front of the workstation clip volume closer to the View Coordinate System (VCS) origin than the near clip plane. In this case the point is returned from the front of the workstation clip volume. This mechanism guarantees that the locator point returned is always in the current clip volume (that is, it is a displayable point).

Since LOCATOR input is returned in World Coordinates (WC), the Device Coordinate (DC) obtained from the above method must be converted to WC. This is done by applying the inverse of the viewing transformation to the DC point to transform it to WC.

Because of the reverse transform required, the view matrix must be invertible (non-singular). If the view matrix is not invertible (singular), all locator points from the view are returned as (0,0,0) and an error is generated. For a matrix to be invertible, its determinant must be non-zero. Most matrixes typically used for view transformations are non-singular. Therefore, if your application generates a matrix which is not invertible (singular), you should check the application carefully to be sure that the intended matrix is being used.

Establishing the Operating Mode of a LOCATOR Device

Your application selects an Operating Mode using the Set Locator Mode **(GPLCMO)** subroutine. **GPCHMO** lets your application specify whether the LOCATOR device is in REQUEST, SAMPLE, or EVENT mode. It also lets your application define the Operating Mode of each LOCATOR device, independently. For example, your application might put one LOCATOR device in EVENT mode and another in SAMPLE mode. The default mode is REQUEST.

Providing Initial Values for a LOCATOR Device

Your application initializes the LOCATOR device through the Initialize Locator **(GPINLC)** subroutine. For example, your application can initialize such things as the type of prompt and the location of the prompt on the display.

The locator echo types include:

- Workstation-dependent Prompts and Echoes
- Small Cross Hair
- Cross Hair
- Rubber Band Line
- Rubber Band Rectangle
- Structure Drag

The locator echo area defines both the region of the display within which the device is active and to which the echo is clipped. This can be used as an effective means of feedback to the operator. When the LOCATOR is activated, the echo is displayed at the initial position specified in the **GPINLC** subroutine.

The LOCATOR device is active only in this echo area. For example, when using the drag structure echo type on the IBM 5080 workstation, the operator may not drag the structure outside the echo area.

Input device initialization subroutine calls may be called only when the device is in REQUEST mode, that is, when the device is not activated.

Obtaining Input From a LOCATOR Device

The subroutine call needed to obtain input from a locator device depends on the operating mode of that device.

Obtaining Input from a LOCATOR Device in REQUEST Mode: In REQUEST mode, your application is responsible for initiating all interactions with the LOCATOR device operator. If your application needs LOCATOR input from an operator before it can proceed, it asks the operator for input by issuing a Request Locator **(GPRQLC)** subroutine.

After issuing the request, the graPHIGS API waits for the operator to satisfy the request. As with all locator input, the returned information indicates the position of the locator prompt in World Coordinates (WC) and the view which was used to transform the screen position back to World Coordinates.

After the requested information is obtained, the system returns control to your application so it can resume processing and determine how to use the returned information.

Obtaining Input From a LOCATOR Device in SAMPLE Mode: Setting a LOCATOR service in SAMPLE mode activates the locator device.

In order to sample a LOCATOR device, your application issues a Sample Locator **(GPSMLC)** subroutine. After issuing a **GPSMLC** subroutine, your application receives the current value of the LOCATOR as fast as the system can supply it.

Obtaining Input From a LOCATOR Device in EVENT Mode: Setting a LOCATOR device in EVENT mode activates the device and establishes an indirect relationship between your application and a LOCATOR device that is mediated by the event queue.

An operator triggers LOCATOR events which are placed on the event queue. Your application can then get that input from the event queue. This lets operator input and application processing occur asynchronously.

An operator can, by triggering the LOCATOR device, send the cursor's location to the event queue.

When the LOCATOR event becomes the next event on the queue, your application issues a **GPAWEV** subroutine. The graPHIGS API gives your application the identification of the device corresponding to the current event report. If the input came from a LOCATOR device and your application wants to use that input, your application issues a Get Locator **(GPGTLC)** subroutine. GPGTLC returns the value of the LOCATOR input.

PICK Device

A PICK device provides a pick status, a pick path, and a pick path depth. The pick path uniquely identifies the picked primitive and its location in the structure network. The cursor controller is an example of a PICK mechanism.

An example of a pick echo is intensification of an individual primitive using the current input echo color. The default echo color is white.

For polygons with the Edge Flag is OFF, a boundary (solid line) is drawn using the echo color. If the Edge Flag is ON, the edge of the polygon is drawn.

Detectability

In order to pick a primitive, that primitive must have visibility and detectability attribute values of "on" Recall that the detectability of each primitive is controlled in a manner like highlighting and visibility. For details refer to Highlighting, Detectability, and Invisibility.

Due to your application's ability to construct a hierarchy with multiple instances of structures, one primitive may appear several times on the view surface. In order to uniquely identify which instance of a primitive is

picked, the system returns a list with PICK identifiers, structure names, and element pointer values. This list represents a path through the structure network. With this information, your application can determine exactly which instance of a structure element was picked.

Suppose an operator needs to pick the right-front wheel on a car. Your application defines each part of the car as detectable. In this scenario, the operator picks the right-front wheel. The system supplies your application with a pick path representing the right-wheel's position in the hierarchy. The following figure depicts the pick path from the **Wheel** structure up to the **Car** structure:

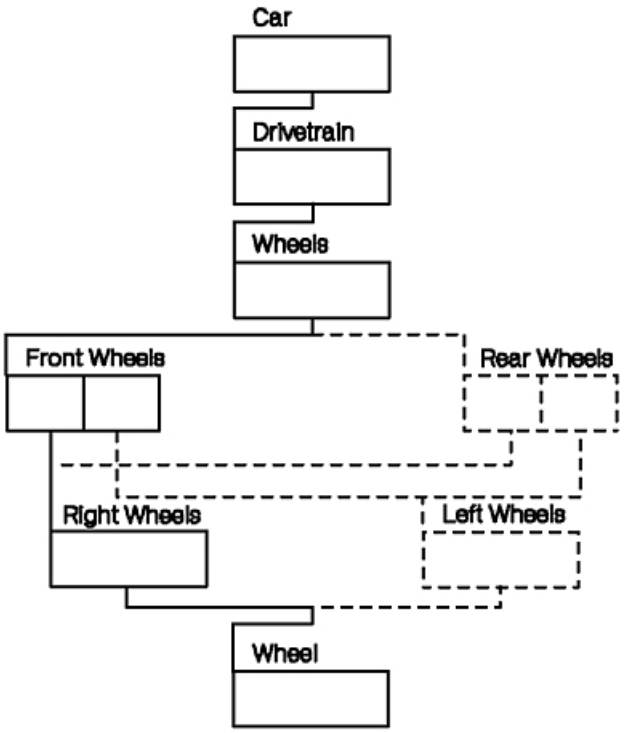


Figure 31. Sample Pick Path for Right-Front Wheel. This illustration shows how a hierarchy is used to determine the pick path. At the top level the Car structure links to the Drivetrain structure. This structure in turn links to the Wheels structure. The Wheels structure has a link to the front wheels structure and a link to the rear wheels structure. The Front Wheels structure has a link to the Right Wheels structure and another link to the Left Wheels structure. The Rear Wheels structure has the same links the Front Wheels structure has. The Right Wheels structure contains a link to the Wheel structure. The Left Wheels Structure similarly links to the Wheels structure.

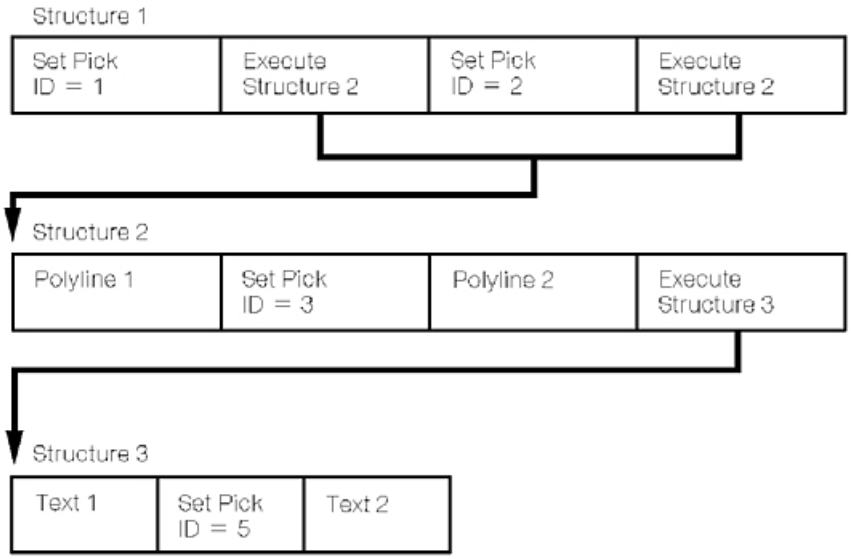
In this case, the system reports that the path from the right-front wheel structure to the root structure passes through the following structures:

- Wheel
- Right Wheels
- Front Wheels
- Wheels
- Drivetrain
- Car

Notice that the pick path contains six entries, making its depth equal to six. Also, the returned information is in a bottom-up fashion, as specified in the Initialize Pick **(GPINPK)** subroutine discussed in Providing Initial Values for a PICK Device. A top-down pick path would have returned the identical pick path information in reverse order.

The graPHIGS API also provides a means of grouping primitives or several primitives with a type of label called a `PICK` identifier. This enables the application program to keep track of primitives without needing to know their element number.

The following figure depicts the returned pick path information:



Case A: Pick Polyline 1 (from bottom up and through the first execute structure 2 call)

Structure ID	Pick ID	Structure Element Number
2	1	1
1	1	2

Case B: Pick Polyline 2 (from bottom up and through the first execute structure 2 call)

Structure ID	Pick ID	Structure Element Number
2	3	3
1	1	2

Case C: Pick Text 1 (from bottom up and through the second execute structure 2 call)

Structure ID	Pick ID	Structure Element Number
3	3	1
2	3	4
1	2	4

Figure 32. Sample Pick Path Information. This illustration shows the returned pick path information given various cases and a defined structure. Structure 1 is defined as a set Pick ID=1, execute structure 2, set Pick ID=2 and another execute structure 2. Structure 2 is defined as a Polyline 1 primitive, set Pick ID=3, Polyline 2 primitive, and an execute structure 3 call. Structure 3 is defined as a Text 1 primitive, set Pick ID=5, and a Text 2 primitive. In Case A, a pick of Polyline 1 occurs. The results are as follows starting from the bottom of the structure up and through the first execute structure 2 call. The first result is a structure ID of 2, a pick ID of 1, and a structure element number of 1. The next result is a structure ID of 1, a pick ID of 1, and a structure element number of 2. In Case B, a pick of Polyline 2 occurs. The results are as follows beginning with the bottom of the structure up and through the first execute structure. The first result gives a structure ID of 2, a pick ID of 3, and a structure element number of 3. The second result provides a structure ID of 1, a pick ID of 1, and a structure element of 2. In Case 3, a pick of Text 1 occurs. The results are as follows starting from the bottom up and through the second execute structure 2 call. The first result returns a structure ID of 3, a pick ID of 3, and a structure element number of 1. The second result provides a structure ID of 2, a pick ID of 3, and a structure element number of 4. The third result returns a structure ID of 1, a pick ID of 2, and a structure element number of 4.

This information includes the:

- structure identifier of each structure in the pick path

- pick identifier active for each structure in the pick path at the specified element in that structure
- structure element number at each level in the pick path

The Request Pick **(GPRQPK)** subroutine enables your application to request pick input.

Establishing the Operating Mode of a PICK Device

Your application selects an operating mode using the Set Pick Mode **(GPPKMO)** subroutine. GPPKMO lets your application specify whether the PICK device is in REQUEST, SAMPLE, or EVENT mode. The Operating Mode of each PICK device is set independently. For example, your application might put one PICK device in EVENT mode and another in SAMPLE mode.

Providing Initial Values for a PICK Device

Your application initializes the PICK device through the Initialize Pick **(GPINPK)** subroutine. For example, your application can initialize such things as the pick echo area and the direction in which the pick path is constructed (top-down or bottom-up).

The pick echo area defines the region of the screen within which the PICK device is active. As with other input devices, the pick prompt and echo are clipped to the pick echo area.

A primitive is picked when it intersects a PICK device's pick aperture. In the graPHIGS API, a pick aperture is a rectangular area defined in Device Coordinates (DC) which is attached to the screen cursor.

Input device initialization subroutines may be called only when the device is in REQUEST mode, that is, when the device is not activated.

Obtaining Input From a PICK Device

The subroutine call needed to obtain input from a PICK device depends on the Operating Mode of that device.

Obtaining Input from a PICK Device in REQUEST Mode: In REQUEST mode, your application is responsible for initiating all interactions with the PICK device operator. If your application needs PICK input from an operator before it can proceed, it asks the operator for input by issuing a Request Pick **(GPRQPK)** subroutine.

After issuing the request, the graPHIGS API waits for the operator to satisfy the request.

As with all PICK input, the returned information contains the status of the pick, the depth of the pick path, and the pick path itself.

After the requested information is obtained, the system returns control to your application so it can resume processing and determine how to use the pick path information.

Obtaining Input from a PICK Device in SAMPLE Mode: Setting a PICK device in SAMPLE mode activates the PICK device.

In order to sample a PICK device, your application issues a Sample Pick **(GPSMPK)** subroutine. After issuing a **GPSMPK** subroutine, your application receives the input information as fast as the system can supply it. Lacking synchronization, the use of a PICK device in this mode has limited utility.

Obtaining Input from a PICK Device in EVENT Mode: Setting a PICK device in EVENT mode activates the device and establishes an indirect relationship between your application and a PICK device that is mediated by the event queue.

An operator triggers PICK events which are placed on the event queue. Your application can then get that input from the event queue. This lets operator input and application processing occur asynchronously.

Upon triggering the PICK device, the depth of the pick and pick path data are added to the event queue.

When the PICK event becomes the next event on the queue, your application issues a **GPAWEV** subroutine call. The graPHIGS API gives your application the identification of the device corresponding to the current event report. If the input came from a PICK device and your application wants to use that input, your application issues a Get Pick **(GPGTPK)** subroutine call. **GPGTPK** returns the value of the PICK input.

The graPHIGS API also supports extended pick subroutine calls which are discussed in Part Two of this book.

STROKE Device

A STROKE input device provides a sequence of positions in World Coordinate (WC), and the view index whose matrix was used to convert the positions from Device Coordinates (DC) to the World Coordinate System (WCS). For a given sequence of points, the active view of highest priority containing all of those points is used for this conversion. Generally, all STROKE input occurs from View 0, the default highest priority view, unless this relationship is changed. The tablet with stylus is an example of a STROKE input device. A STROKE input value is simply a sequence of location values.

Establishing the Operating Mode of a STROKE Device

Your application selects an Operating Mode of a STROKE device using the Set Stroke Mode **(GPSKMO)** subroutine. **GPSKMO** lets your application specify whether the STROKE device is in REQUEST, SAMPLE, or EVENT mode. It also lets your application define the Operating Mode of each STROKE device, independently. For example, your application might put one STROKE device in EVENT mode and another in SAMPLE mode.

Providing Initial Values for a STROKE Device

Your application initializes the STROKE device through the Initialize Stroke **(GPINSK)** subroutine. For example, your application can specify such things as the number of points in the initial stroke, the coordinates of each point in the initial stroke, the type of prompt and echo, and the length of the STROKE input buffer.

The stroke echo area defines the region of the screen within which the STROKE device is active. As with other input devices, the stroke prompt and echo are clipped to the stroke echo area.

GPINSK lets your application provide an initial sequence of STROKE locations. STROKE input values are added to the STROKE buffer beginning at a location specified in the **GPINSK** subroutine. This location is referred to as the editing position within the STROKE buffer. For example, an editing position of one causes the replacement of the initialized STROKE values beginning at the first point. An editing position of one greater than the number of initial points appends the STROKE input to the initialized values.

The graPHIGS API enables your application to specify the size of the STROKE buffer, which is the maximum number of points allowable for STROKE input. Notice that this may be limited by the maximum buffer size on a given workstation.

STROKE devices have different echo types including:

- Polyline with its attributes
- Polymarker with its attributes

Input device initialization subroutines may be called only when the device is in REQUEST mode, that is, when the device is not activated.

Obtaining Input from a STROKE Device

The subroutine call needed to obtain input from a STROKE device depends on the Operating Mode of that device.

Obtaining Input from a STROKE Device in REQUEST Mode: In REQUEST mode, your application is responsible for initiating all interactions with the STROKE device operator. If your application needs STROKE input from an operator before it can proceed, it asks the operator for input by issuing a Request Stroke **(GPRQSK)** subroutine call.

After issuing the request, the graPHIGS API waits for the operator to satisfy the request.

When the operator triggers the STROKE device, the system sends your application the number of coordinate points in the array, the array containing the points, and the view in which the stroke occurred.

After the requested information is obtained, the system returns control to your application so it can resume processing and determine how to use the returned information.

Obtaining Input from a STROKE Device in SAMPLE Mode: Setting a STROKE device in SAMPLE mode activates the STROKE device.

In order to sample a STROKE device, your application issues a Sample Stroke **(GPSMSK)** subroutine call. After issuing a **GPSMSK** subroutine, your application receives, from the API, the input information as fast as the system can supply it. Lacking synchronization, the use of a STROKE device in this mode has limited utility.

Obtaining Input from a STROKE Device in EVENT Mode: Setting a STROKE device in EVENT mode activates the device and establishes an indirect relationship between your application and a STROKE device that is mediated by the event queue.

An operator triggers STROKE events which are placed on the event queue. Your application can then get that input from the event queue. This lets operator input and application processing occur asynchronously.

When the STROKE event becomes the next event on the queue, your application issues a **GPAWEV** subroutine call. The graPHIGS API gives your application the identification of the device corresponding to the current event report. If the input came from a STROKE device and your application wants to use that input, your application issues a Get Stroke **(GPGTSK)** subroutine. **GPGTSK** returns the value of the STROKE input.

STRING Device

A STRING device allows an operator to input text. The keyboard is an example of a STRING input mechanism. To input a string of characters, the operator presses alphanumeric keys. To trigger a typical STRING device, the operator presses the ENTER key.

Establishing the Operating Mode of a STRING Device

Your application selects an Operating Mode using the Set String Mode **(GPSTMO)** subroutine. **GPSTMO** lets your application specify whether the STRING device is in REQUEST, SAMPLE, or EVENT mode. It also lets your application define the Operating Mode of each STRING device, independently. For example, your application might put one STRING device in EVENT mode and another in SAMPLE mode.

Providing Initial Values for a STRING Device

Your application initializes the STRING device through the Initialize String **(GPINST)** subroutine. For example, your application can specify such things as the length of an initial string, the text of the initial string and the maximum length of the STRING input buffer.

The string echo is positioned in the lower left corner of the echo area specified in the **GPINST** subroutine. String echo is clipped to this area.

GPINST lets your application provide an initial string which is placed in the STRING buffer. String input values are added to the string buffer beginning at a location specified in the **GPINST** subroutine. This position is referred to as the initial cursor position. For example, a cursor position of one causes

replacement of the initialized string values, beginning at the first character. A cursor position of one greater than the number of initial characters appends string input to the initialized values.

The graPHIGS API enables your application to specify the size for the STRING input buffer, which is the maximum number of characters allowable for string input. This may be limited by the maximum supported buffer size on a given workstation.

The STRING device echo typically is a string of displayed characters. Notice that your application can prevent the characters from displaying by turning echo OFF.

String Echo Type 2 allows your application to pass a protected prompt string to a STRING device on the Initialize String **(GPINST)** subroutine. This prompt string is passed in via the data record and is displayed in the lower left corner of the echo area, and is followed by the input buffer echo. The prompt may not be typed over and is not returned with the STRING device input. The initial cursor position is given as an offset in the input buffer. The specified STRING input buffer size, plus the prompt length, must be less than or equal to the maximum STRING input buffer size as defined in the actual WDT.

Input device initialization functions may be called only when the device is in REQUEST mode, that is, when the device is not activated.

Obtaining Input from a STRING Device

The subroutine call needed to obtain input from a STRING device depends on the Operating Mode of that device.

Obtaining Input from a STRING Device in REQUEST Mode: In REQUEST mode, your application is responsible for initiating all interactions with the STRING device operator. If your application needs STRING input from an operator before it can proceed, it asks the operator for input by issuing a Request String **(GPRQST)** subroutine call.

After issuing the request, the graPHIGS API waits for the operator to satisfy the request. When the operator triggers the STRING device, the system sends your application the length of the string in bytes and the text string. After the requested information is obtained, the system returns control to your application so it can resume processing and determine how to use the returned information.

Obtaining Input from a STRING Device in SAMPLE Mode: Setting a STRING device in SAMPLE mode activates the STRING device.

In order to sample a STRING device, your application issues a Sample String **(GPSMST)** subroutine call. After issuing a **GPSMST** subroutine, your application receives the string data as fast as the system can supply it. Lacking synchronization, the use of a STRING device in this mode has limited utility.

Obtaining Input from a STRING Device in EVENT Mode: Setting a STRING device in EVENT mode activates the device and establishes an indirect relationship between your application and a STRING device that is mediated by the event queue.

An operator triggers STRING events which are placed on the event queue. Your application can then get that input from the event queue. This lets operator input and application processing occur asynchronously.

When the STRING event becomes the next event on the queue, your application issues a **GPAWEV** subroutine. The graPHIGS API gives your application the identification of the device corresponding to the current event report. If the input came from a STRING device and your application wants to use that input, your application issues a Get String **(GPGTST)** subroutine call. **GPGTST** returns the value of the STRING input.

VALUATOR Device

A VALUATOR device provides a real number. A dial is an example of a VALUATOR input mechanism. Each dial on the workstation is a separate VALUATOR device. For example, the IBM 5080 can have eight VALUATOR devices.

Modified Sample Program 1

You can now add a VALUATOR device to the sample program. The VALUATOR will allow you to modify the view matrix of one of the views and provide a zooming effect on the house. Modify the sample program as follows:

1. In the DECLARE VARIABLES section add the following statements:

```
INTEGERx4 VDATAL,VDATA(4)
REALx4 VVALUE,LOW,HIGH,SCALE(2),VMATRX(9)
DATA VDATAL /16/
DATA VDATA /1,0,0,10/
DATA LOW /0.5/
DATA HIGH /2.0/
```

2. In the INPUT SUBROUTINES section, add the following statement before the CALL GPAWEV statement:

```
CALL GPINVL(WSID,1,1.,4,AREA,LOW,HIGH,VDATAL,VDATA)
CALL GPVLMO(WSID,1,3,2)
```

3. Also in the INPUT SUBROUTINES section, add the following statements between the second ENDIF and GOTO 100 statements:

```
IF(CLASS.EQ.3) THEN
  IF(DEVICE.EQ.1) THEN
    CALL GPGTVL(VVALUE)
    SCALE(1) = VVALUE
    SCALE(2) = VVALUE
    CALL GPSC2(SCALE,VMATRX)
    CALL GPXVR(WSID,VIEW1,18,VMATRX)
    CALL GPUPWS(WSID,2)
  ENDIF
ENDIF
```

Modify your sample program, compile it and run it. Try turning dial 1 of the VALUATOR, there should be a zooming effect on the house on the lower left view. Also, the echo of the VALUATOR should be displayed on the lower left corner of the screen.

Establishing the Operating Mode of a VALUATOR Device

Your application selects an Operating Mode using the Set Valuator Mode **(GPVLMO)** subroutine call. **GPVLMO** lets your application specify whether the VALUATOR device is in REQUEST, SAMPLE, or EVENT mode. It also lets your application define the Operating Mode of each VALUATOR device, independently. If, for example, your application is interacting with a set of dials, the system defines each dial separately. This lets your application set one dial in EVENT mode and another in SAMPLE mode.

In the sample program, the statement:

```
CALL GPVLMO(WSID,1,3,2)
```

sets dial 1 to EVENT mode.

Providing Initial Values for a VALUATOR Device

Your application initializes the VALUATOR device through the Initialize Valuator **(GPINVL)** subroutine. For example, your application can specify such things as the VALUATOR's initial value, its minimum and maximum values, and where the VALUATOR's echo is placed.

In the sample program, the statement:

```
CALL GPINVL(WSID,1,1.,4,AREA,LOW,HIGH,VDATAL,VDATA)
```

initializes dial 1 of the VALUATOR device.

One valuator echo type is a digital display. The digital display changes as the operator moves the valuator. If the VALUATOR device is a dial, your application can specify that one rotation of the dial covers the entire range, or that a number of rotations covers the range. In the sample, the VDATA parameter establishes 10 turns between the LOW and HIGH values. The initial value of the dial is 1.0.

The valuator echo is positioned in the lower left corner of the echo area specified in the **GPINVL** subroutine call. The valuator echo is clipped to this area.

Input device initialization functions may be called only when the device is in REQUEST mode, that is, when the device is not activated.

Obtaining Input from a VALUATOR Device

The subroutine call needed to obtain input from a VALUATOR device depends on the Operating Mode of that device.

Obtaining Input from a VALUATOR Device in REQUEST Mode: In REQUEST mode, your application is responsible for initiating all interactions with the VALUATOR device operator. If your application needs VALUATOR input from an operator before it can proceed, it asks the operator for input by issuing a Request Valuator **(GPRQVL)** subroutine.

After issuing the request, the graPHIGS API waits for the operator to satisfy the request.

When the operator triggers the VALUATOR device, the system sends that value to your application.

After the requested information is obtained, the system returns control to your application so it can resume processing and determine how to use the returned information.

Obtaining Input from a VALUATOR Device in SAMPLE Mode: Setting a VALUATOR device in SAMPLE mode activates the valuator device.

In order to sample a VALUATOR device, your application issues a Sample Valuator **(GPSMVL)** subroutine. After issuing a **GPSMVL** subroutine, your application receives, from the graPHIGS API, the VALUATOR data as fast as the system can supply it.

Note: VALUATORS in EVENT mode typically provide a more efficient communication mechanism with your application.

Obtaining Input from a VALUATOR Device in EVENT Mode: Setting a VALUATOR device in EVENT mode activates the device and establishes an indirect relationship between your application and a VALUATOR device that is mediated by the event queue.

An operator triggers VALUATOR events which are placed on the event queue. On the IBM 5080 workstation, a VALUATOR event is triggered by dial movement. This means that the application is notified of dial values only when they change. Your application can then get that input from the event queue. This lets operator input and application processing occur asynchronously.

When the VALUATOR event becomes the next event on the queue, your application issues a **GPAWEV** subroutine call. The graPHIGS API gives your application the identification of the device corresponding to the current event report. If the input came from a VALUATOR device and your application wants to use that input, your application issues a Get Valuator **(GPGTVL)** subroutine call. **GPGTVL** returns the value of the VALUATOR input.

In the sample program, the statement:

```
CALL GPGTVL(VVALUE)
```

gets the value of dial 1 from the event queue. This value is then used to initialize the scaling vector which in turn is used to determine the view matrix VMATRX for the zooming effect. The matrix is calculated using the utility function **GPSC2** The **GPXVR** subroutine is then used to assign VMATRX to view 1 of the WSID workstation. An update workstation operation is then needed.

Chapter 8. Structure Editing

In Chapter 3. Structures you learned how to define structure content. The graPHIGS API also provides powerful subroutine calls that enable your application to edit the content (elements) of those structures. In addition to discussing those subroutines, this chapter discusses the operations that affect entire structures.

As you read this chapter, keep in mind that structure content editing operations require an open structure. Operations that affect entire structures do not require an open structure.

Structure Content Editing

When an application program opens a structure, the element pointer points to the last element of the structure. In an empty structure, the element pointer points to the null element, conceptually element zero.

Your application can add elements one after another, because the graPHIGS API automatically increments the element pointer to each new element.

Inserting Structure Elements

In order to edit an element within a structure, your application must position the element pointer. When processing a subroutine call that creates an element, the system either inserts that element into the structure *after* the current element pointer or replaces the element pointed to by the element pointer. The Set Edit Mode **(GPEDMO)** subroutine controls whether the elements are inserted or replaced. The default Edit Mode is `1=INSERT_MODE` Therefore, to insert an element, your application must position the element pointer at the element immediately *before* the insertion location.

The graPHIGS API provides the Set Element Pointer **GPEP** subroutine to position the element pointer at any element position within a structure.

If your application specifies a negative number or a zero, in the **GPEP** subroutine, the system sets the element pointer at conceptual element zero.

If your application specifies a number in the **GPEP** subroutine that is greater than the last element's number, the system sets the element pointer at the last element.

The following figure depicts a scenario in which your application inserts two label elements into an existing structure. Notice that your application could have chosen to insert any other valid structure element instead.

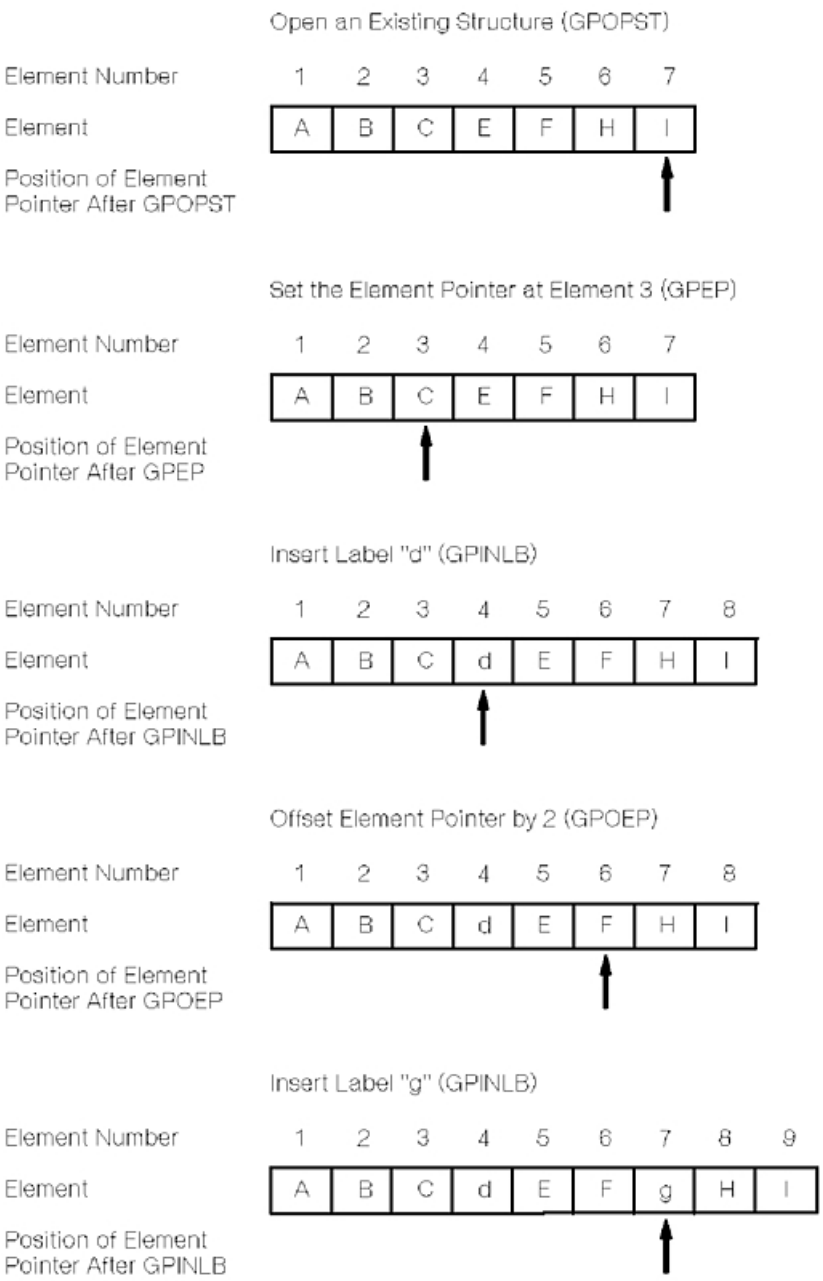


Figure 33. Inserting Elements. This illustration shows a scenario in which your application inserts two label elements into an existing structure.

The graPHIGS API provides the Insert Label **(GPINLB)** subroutine in order to create a label element within a structure. Your application assigns an integer identifier to each label. These integer identifiers do *not* need to be unique.

A label is a type of structure element. It may be used for anything. It has no graphical effect. The graPHIGS API provides labels so your application can more easily edit structure content.

With labels, your application can delineate an element or group of elements, then later find that element or element group by advancing the element pointer to a specified label.

In the figure, *Inserting Elements*, your application opens the structure with the **GPOPST** subroutine. After opening the structure, the current element pointer points to the last element in the structure.

In this scenario, your application uses the **GPEP** subroutine to position the element pointer immediately before the desired insertion point.

Your application then inserts a label “d”, using the **GPINLB** subroutine. After the insertion, the system positions the current element pointer at the inserted element, and renumbers the following elements.

Next, your application uses the Offset Element Pointer **(GPOEP)** subroutine to position the element pointer forward or backward in the structure. The system adds the positive or negative offset value specified in the **GPOEP** subroutine to the current element pointer value. If the resulting value would move the element pointer outside of the structure, the system moves the element pointer to the nearest end of the structure, and issues a warning.

After your application uses the **GPINLB** subroutine to insert another label “g”, the system renumbers the following elements.

Another way to position the element pointer is by using one of the following subroutines:

- Set Element Pointer at Label **(GPEPLB)**
- Set Element Pointer at Pick Identifier **(GPEPPK)**
- Locate Element Pointer at Element Code **(GPEPCD)**

GPEPLB lets your application position the element pointer at any label element within an open structure. **GPEPPK** does the same for `PICK` identifier elements. **GPEPCD** positions the element pointer for any structure element based on the element’s identifier code.

Your application locates a label, `PICK` identifier, or element code by specifying its integer identifier. The system searches for the next label that has the specified integer identifier, starting at the element immediately *after* the current element pointer. When it finds the identifier, it moves the element pointer to that position in the structure.

If the system encounters the end of a structure, it resets the element pointer to the first element and continues searching for the integer identifier until it finds it or reaches the start element.

Structures can contain more than one label with the same integer identifier. When they do, invoking **GPEPLB** advances the element pointer to the next label containing the specified integer identifier. **GPEPPK** operates in an identical fashion.

Deleting Structure Elements

The graPHIGS API lets your application program delete elements or groups of elements in the following ways:

- by deleting an individual element
- by deleting the elements within a specified range
- by deleting the elements between two labels

Whenever your application deletes an element, or group of elements, the system sets the element pointer to the element before the deleted element or group of elements using method one or two. This lets your application add new elements into the deleted location without moving the element pointer. When using method three, your application can leave the element pointer at either label.

Deleting Individual Elements

To delete an individual element, your application sets the element pointer to the desired element, then deletes that element using the Delete Element **(GPDLE)** subroutine. After deletion, the API sets the element pointer to the element preceding the deleted element. The following figure depicts the results of deleting one element:

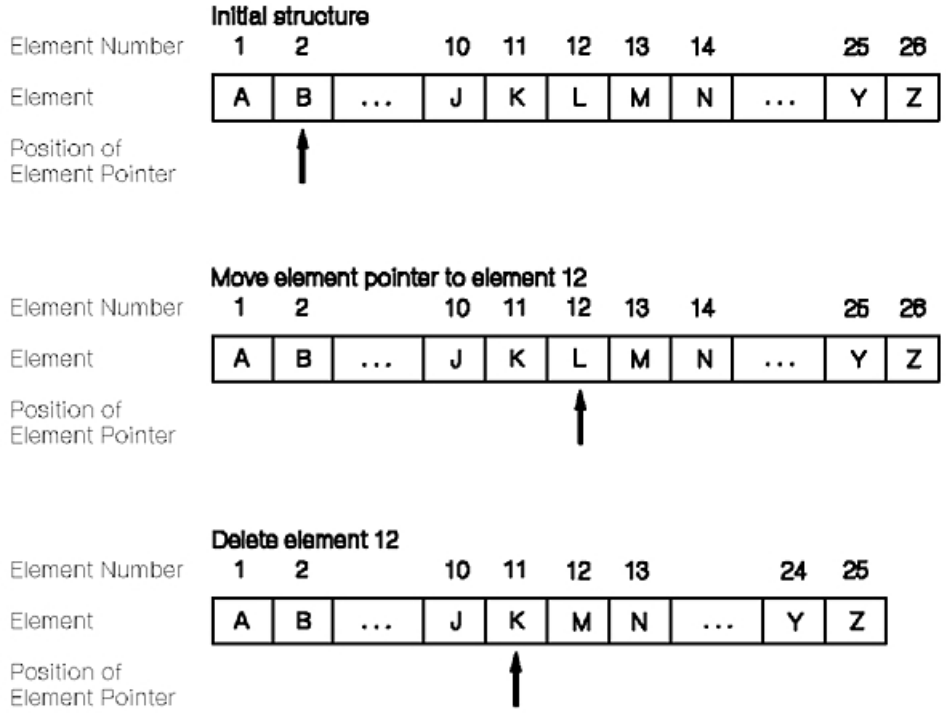


Figure 34. Deleting One Element. This illustration shows the process to delete an individual element.

Deleting Elements within a Specified Range

To delete a range of elements, your application specifies a range of element numbers with the Delete Range **(GPDLER)** subroutine. The system deletes all elements within the range, including the specified range elements.

If either number specified in the **GPDLER** subroutine is outside of the structure's range, the system maps that number to the closest end of the structure. For example, if the structure has 20 elements and your application tries to delete the elements between 12 and 30, the system deletes the elements between 12 and 20.

In addition, your application program can specify the numbers in either order. Deleting elements between 12 and 30 has the same effect as deleting elements between 30 and 12.

After deleting the elements, the system moves the element pointer to the element immediately before the deleted range, regardless of element pointer's previous position.

The following figure depicts the results of deleting a range of elements:

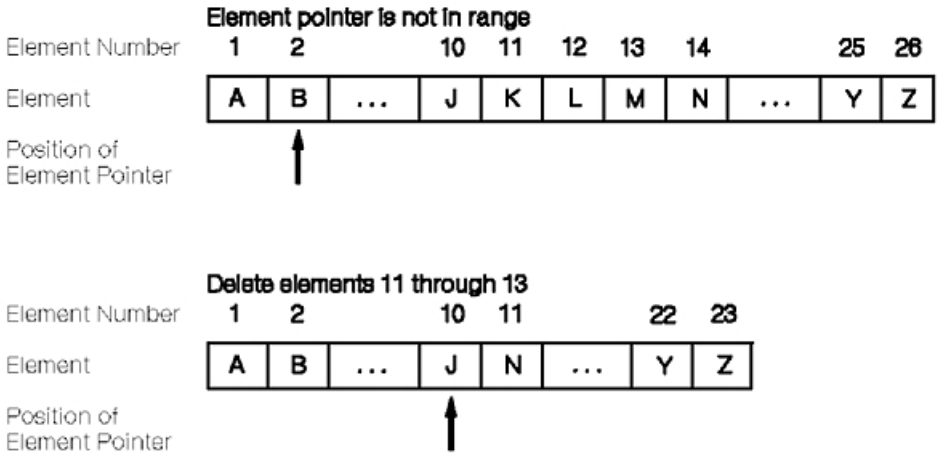


Figure 35. Results of Deleting a Range of Elements. This illustration shows the results of deleting a range of elements.

Deleting Elements between Labels

To delete elements between two labels, your application specifies two labels by using a Delete Element Between Labels **(GPDELB)** or Delete Element Group (**GPDLEG**) subroutines.

When your application issues a **GPDELB** subroutine, the system initiates a search for the first occurrence of label one, starting at the element pointer's current position. If the search reaches the end of the structure, it continues from the beginning of the structure. After finding the first label, the system searches for label two in the same manner, including starting at the *same* original element pointer position. The same label structure element will not match both labels if the same label `id` is specified.

When the system finds both labels, it deletes all elements between, but not including the labels. Notice that the order in which your application specifies the labels affects the position of the element pointer after the deletion. The system resets the element pointer to the *first* label specified in the **GPDELB** subroutine. This allows your application to delete the same range and leave the element pointer in either location.

If the system does not find either label, it reports an error.

The function of the **GPDLEG** subroutine is similar to the **GPDELB** subroutine in that two labels are specified to delineate a group of elements to be deleted. **GPDLEG** differs from **GPDELB** in the following ways:

- The system does not find either label from the start of the search to the end of the structure, an error is generated and the search is discontinued.
- The search for the second label starts at the position of the first label.
- Either, both, or neither of the search labels can optionally be deleted as part of the element group.

The following figure depicts the results of deleting elements between specified labels using the **GPDELB** subroutine. In the figure, elements "j" and "m" are labels.

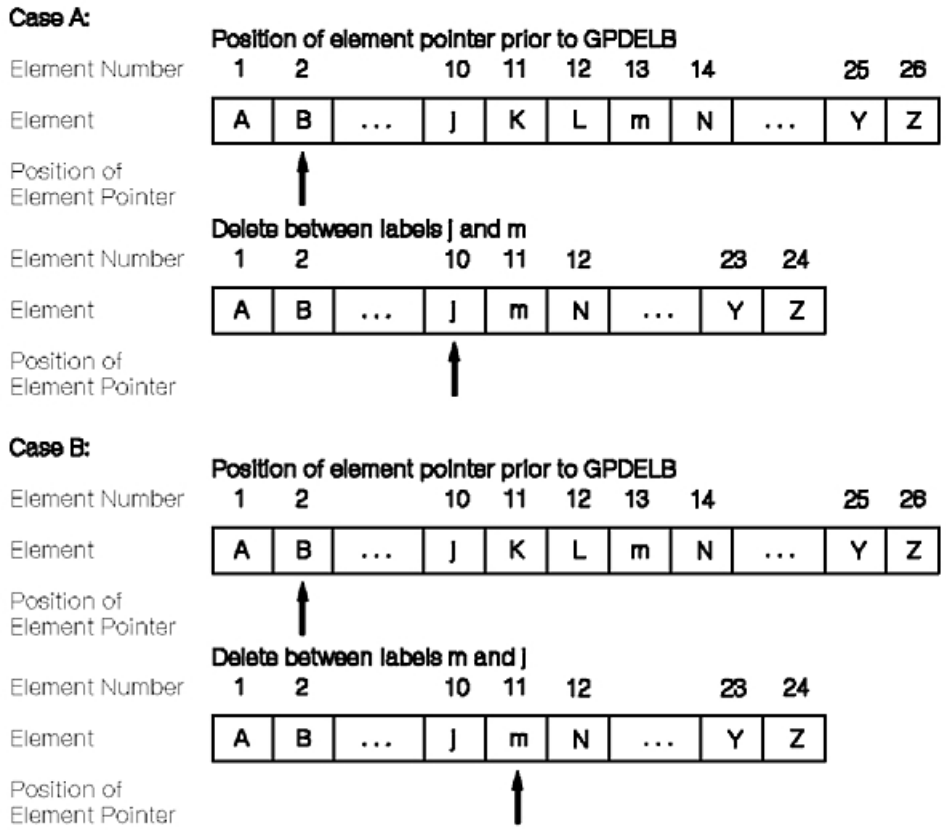


Figure 36. Deleting Elements Between Labels. This illustration shows the results of deleting elements between specified labels using the GPDELB subroutine.

The graPHIGS API lets your application specify duplicate labels within structures. However, you should exercise **caution** when deleting elements between duplicate labels.

Copying and Moving Structure Elements

The Copy Structure **(GPCPST)** subroutine lets your application copy the entire contents of one structure into another. The system copies all of the elements of a specified structure into the currently open structure at the location immediately following the current element pointer. The API then moves the current element pointer to the last inserted element. The following figure depicts how a copy structure instruction from your application is handled. Note that the copied elements are inserted into the structure even if the structure Edit Mode is set to `2=REPLACE_MODE`.

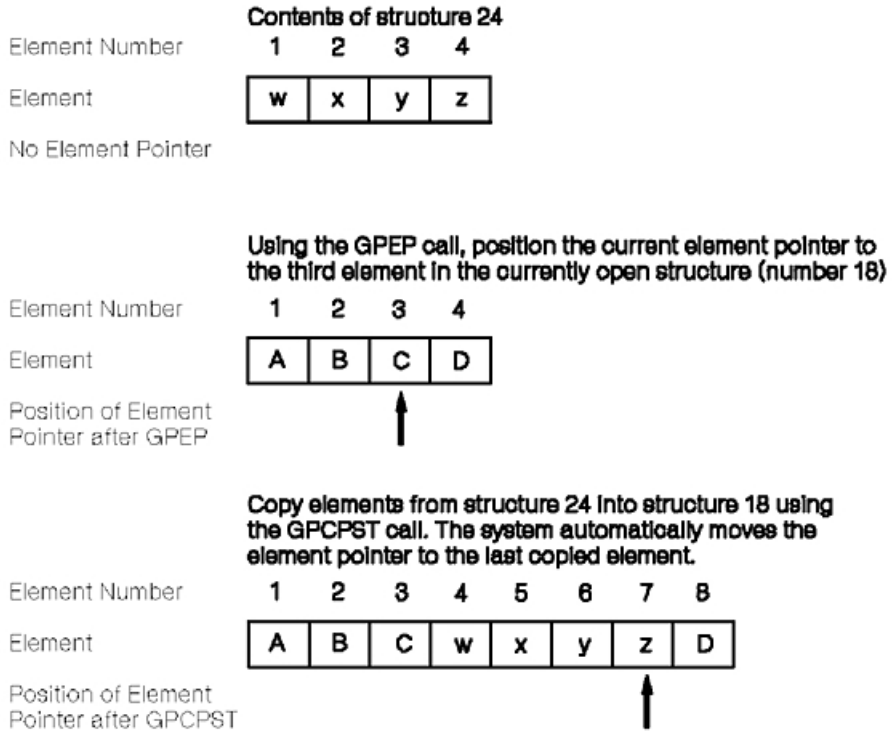


Figure 37. Copying a Structure. This illustration shows how a copy structure instruction from your application is handled.

The Copy Element Range **(GPCPER)** subroutine copies a range of elements from the specified structure, into the open structure following the current element pointer. The API then moves the current element pointer to the last inserted element.

Using the current element pointer, the Move Element Range **(GPMVER)** subroutine lets your application move a range of elements to the the location following the element pointer. The API then moves the current element pointer to the last element that was moved.

Closing Structures

The graPHIGS API lets your application close an open structure. When your application finishes editing the open structure, it *must* close the structure before opening another structure. It closes the currently open structure by using the Close Structure **(GPCLST)** subroutine.

Modified Sample Program 1

Modifying the sample program according to the following statements will enable you to interactively edit the house structure.

1. In the DECLARE VARIABLES section, add the following statements:

   ```
   INTEGERx4 ACTNUM,ACTLEN,CDATA(2),TERM
   ```

2. In this same section, modify the data length and data of the choice input device initialization parameters. First, modify the dimension of the ***variable Data*** You should have DATA(4) Change it to DATA(7) Then modify the DATA statements as follows:

   ```
   DATA DATA /28/
         DATA DATA /4,0,0,2,1,1,2/
   ```

   These changes will light the first and fourth Lighted Program Function Key (LPFK).

3. In this same section, change the DATA statement to indicate that the linetype attribute is set to INDIVIDUAL:

   ```
   DATA ATFLAG /1,1,2/
           to
         DATA ATFLAG /2,1,2/
   ```

4. In the DATA CREATION section, modify the the code which creates the second structure as follows:

   ```
   CALL GPOPST(STRID(2))
         CALL GPMLX2(TRANSD,POST)
         CALL GPINLB(1)
         CALL GPLT(6)
         CALL GPPLCI(6)
         CALL GPPL2(5,2,DOOR)
         CALL GPCLST
   ```

5. In the INPUT SUBROUTINES section, after the statement:

   ```
   IF(CHOICE.EQ.1) GOTO 200
   ```

   add the following statements:

   ```
   IF(CHOICE.EQ.4) THEN
           CALL GPOPST(STRID(2))
           CALL GPEDMO(2)
           CALL GPEPLB(1)
           CALL GPOEP(1)
           CALL GPQED(1,32,ERRIND,ACTNUM,ACTLEN,CDATA,TERM)
           IF(ERRIND.NE.0) GOTO 200
           IF(CDATA(2).EQ.6) CALL GPLT(1)
           IF(CDATA(2).EQ.1) CALL GPLT(6)
           CALL GPCLST
           CALL GPUPWS(WSID,2)
        ENDIF
   ```

After modifying your sample program, compile and run it. The fourth LPFK should be lit. Press it. The Line Type of the door should change from dashed to solid. Press it again. It should change back to dashed.

The instructions you have added modify the door structure by inquiring the structure element type. The application first opens the structure, then editing mode is set to replace, and then the pointer is located at the label. The pointer is then moved one element down and placed where the Line Type element is. Now the application inquires the element type, SOLID_LINE or DASHED, and replaces the element according to the result of the inquire. The structure is then closed and the workstation updated.

Operations on Entire Structures

This section discusses operations that affect entire structures. These operations do not require an open structure. In fact, only the empty structure operation can be done in either the structure open or closed state. For your application to delete structures, the system must be in the structure closed state.

The following figure depicts a structure network:

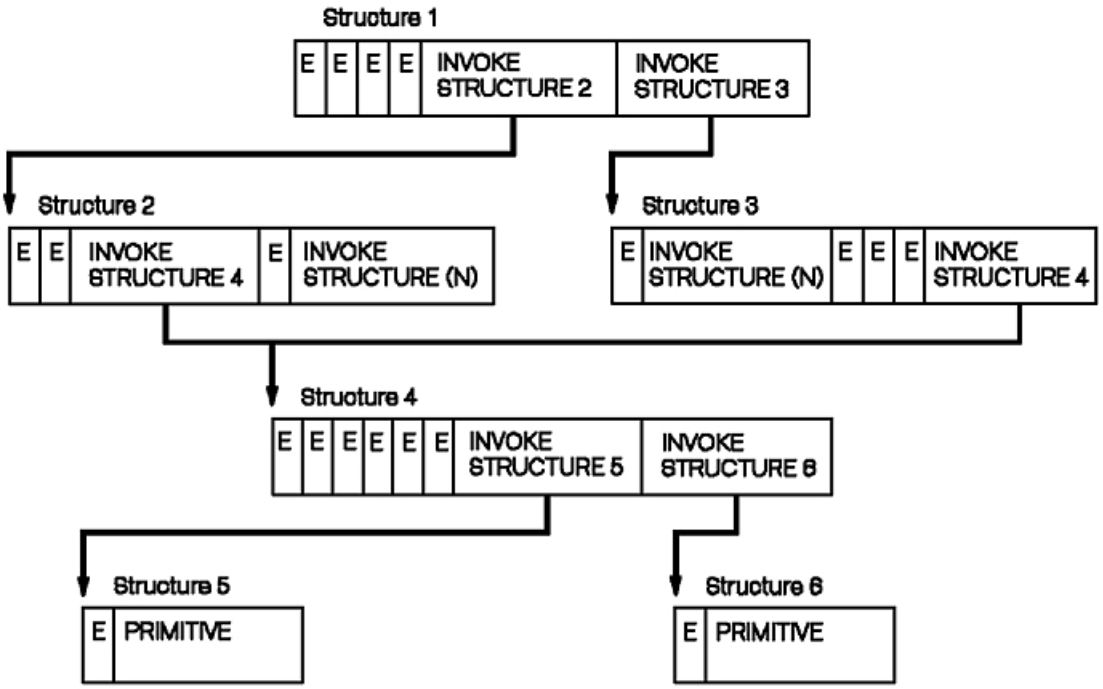


Figure 38. Sample Structure Network Before Empty and Delete Operations. This diagram shows a network of six structures.

The following figure depicts the results of an empty operation applied to the structure network in the figure, Sample Structure Network Before Empty and Delete Operations:

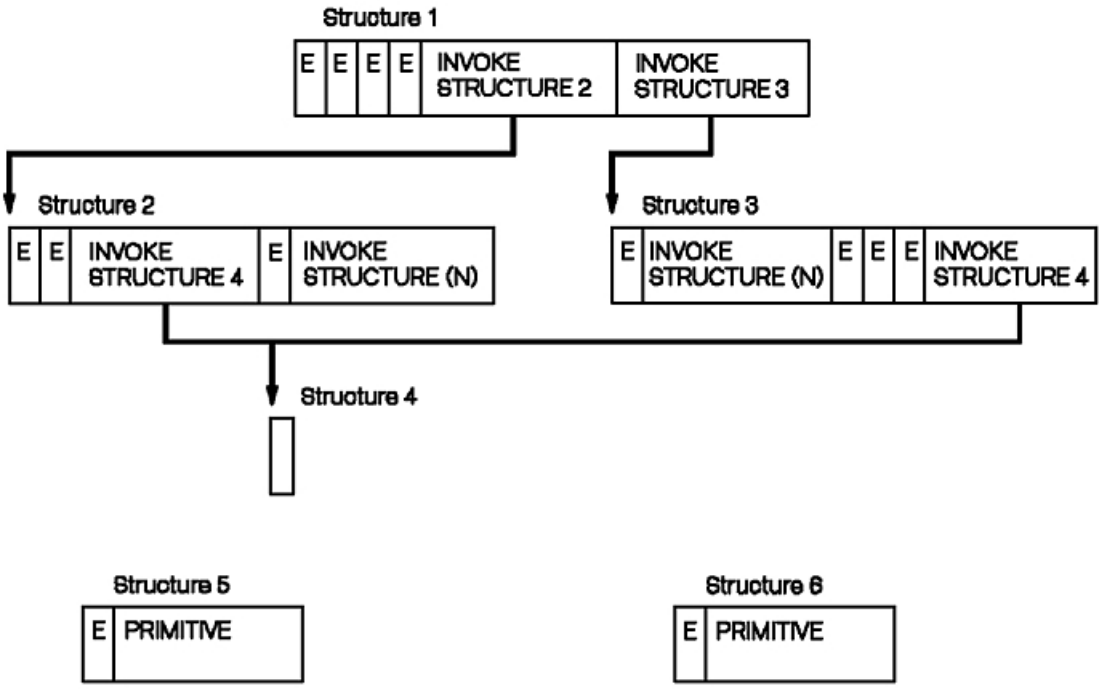


Figure 39. Results of an Empty Structure (GPEST) subroutine. This illustration shows the results of an empty operation applied to the structure network.

The following figure depicts the results of a delete operation applied to the structure network in the figure, Sample Structure Network Before Empty and Delete Operations:

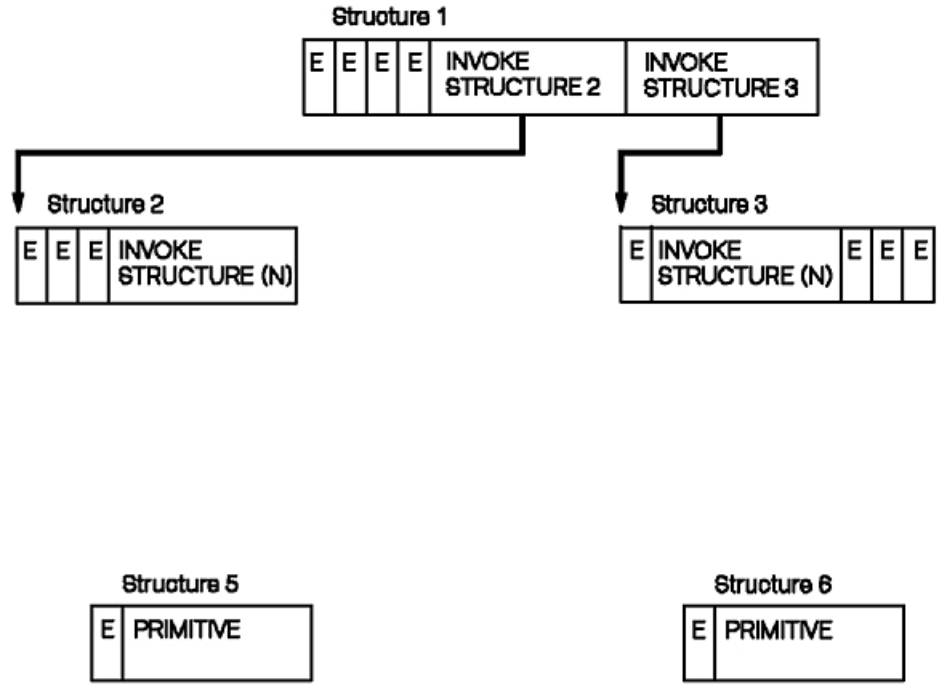


Figure 40. Results of a Delete Structure (GPDLST) subroutine. This illustration shows the results of a delete operation applied to the structure network.

Emptying a Structure

Your application can delete a structure's content by issuing the Empty Structure **(GPEST)** subroutine. This lets your application empty open or non-opened structures. If the emptied structure is open, the graPHIGS API resets the element pointer to the conceptual element zero.

The figure, *Results of an Empty Structure (GPEST) subroutine*, depicts the results of emptying structure 4. Notice that this does *not* remove the reference to structure 4 from structures 2 and 3. However, it does disconnect structures 5 and 6 from structures 1, 2, 3, and 4.

Deleting Structures

The graPHIGS API enables your application to delete an individual structure, a group of structures, or all structures.

The Delete Structure **(GPDLST)** subroutine removes all references to the identified structure, since a structure exists by reference. The figure, *Results of a Delete Structure (GPDLST) subroutine*, depicts the results of a Delete Structure subroutine call. Notice that the deletion of structure 4 also implies removing all references to structure 4. Notice also that although structures 5 and 6 still exist, they are no longer connected to structures 1, 2, and 3. Conceptually, deleting an individually identified structure involves deleting "upward" in the hierarchy and also deleting the elements that reference the identified structure.

The Delete Structure Network **(GPDLNT)** subroutine deletes a specified structure and all structures it references. These include:

- all structures referenced directly in the element list of the specified structure
- all structures referenced indirectly as part of the network emanating from the specified structure

The following figure shows the results of deleting a structure network—again structure number 4 from the figure, *Sample Structure Network Before Empty and Delete Operations*. Notice that the deletion of structure 4 removes not only the references to structure 4, but also the network of all structures that comprise structure 4, including structures 5 and 6.

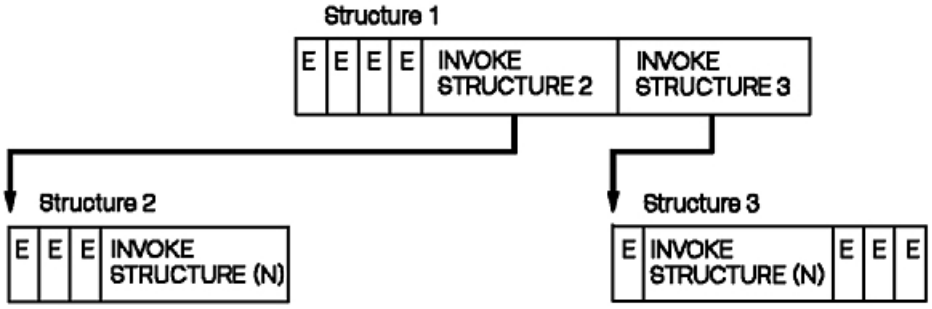


Figure 41. Results of a Delete Structure Network (GPDLNT) subroutine. This illustration shows the results of deleting a structure network.

The Delete Structure Network Conditionally **(GPDLNC)** subroutine deletes a specified root structure and conditionally deletes all of the structures referenced, both directly and indirectly, by the root structure. Only those structures which are not referenced directly or indirectly by other root structures will be deleted. For example, suppose your application creates the following structure network:

- structure **A** is the root structure which executes structure **B**
- structure **B** executes structure **D**
- structure **C** is a root structure which also executes structure **D**

If your application invokes **GPDLNC** to conditionally delete the structure network originating at structure **A**, only structures **A** and **B** will be deleted. Structure **D** will not be deleted because structure **C**, which is not part of structure **A**'s network, also references structure **D**.

The Delete All Structures **(GPDAST)** subroutine causes the system to delete all structures, their identifiers, and their contents.

Chapter 9. Inquiry Subroutines

A device-independent Application Programming Interface (API) *does not* guarantee that all programs run identically on all supported devices. A device-independent API provides the tools that allow properly written application programs to run identically on all supported devices.

In order for your applications to run on other workstations in the future, your application should adapt to a workstation's capabilities. That is why the key to creating device-independent programs involves the proper use of inquiries to determine the specific capabilities of a system and its workstations.

Device-independent applications need not perform any differently than device-specific applications. You must initially invest more effort in program development. The payback is the reduction of effort in migrating the application to other supported graphics devices. If programmed properly, your application can run unmodified on other than the original target workstation.

Inquiries enable your application to determine:

- the capabilities of the system
- the state of the system
- information about a workstation's capabilities
- information about a workstation's state
- information about structures and their content

This chapter discusses *categories* of inquiries. It does not cover every inquiry subroutine call. It presents some important, commonly used inquiries. Refer to *The graPHIGS Programming Interface: Subroutine Reference* for more information about each inquiry subroutine. Refer to *The graPHIGS Programming Interface: Technical Reference* for the content of state lists and description tables.

System Related Inquiries

System related inquiries let your application inquire into the system's operating states, state lists, and description table. System related inquiries can be used to inquire information such as:

- the maximum number of simultaneously open workstations
- the current operating states of the system such as the workstation state and the structure state
- the currently open structure's identifier
- the current element pointer value
- whether there are more simultaneous events on the input queue
- supported text fonts and their characteristics
- information about the workstation and views to which a structure is associated

Workstation Related Inquiries

The purpose of workstation inquiries is to enable your application to inquire the state of a workstation, and adapt to its capabilities, such as differing screen sizes, number of supported colors, and supported input devices.

In order to determine such workstation-dependent capabilities, your application queries a workstation's description table (WDT) and a workstation's state list (WSL).

Workstation Description Table

All inquiries whose first parameter is a workstation type inquire information from the WDT corresponding to that workstation type. The values in the WDT are provided by and initialized by the system. It contains all of the workstation's defaults. It also contains all of the general characteristics of that workstation's type.

Certain values in the WDT are configuration variable. These values depend on the type and configuration of each workstation. When the graPHIGS API opens a particular workstation, it obtains the necessary device-specific information, then inserts that information and creates an actual WDT for that workstation.

Each actual WDT is identified by a unique system generated workstation type which can be inquired using the Inquire Realized Connection and Type **(GPQRCT)** subroutine. **GPQRCT** takes a workstation identifier as its' first parameter and returns both the actual connection identifier and workstation type for the workstation. The actual workstation type can then be used in any WDT inquiry to determine the actual capabilities of the workstation.

WDT inquiries can be used to inquire information such as:

- default bundle table definitions
- the extent of support for the graPHIGS API output primitive attributes
- default input device support

In the sample program, the statements:

```
CALL GPQRCT (WSID, ILEN, ERRIND, OLEN, ACONID, AWSTYP)
CALL GPQDS (AWSTYP, ERRIND, UNITS, CSIZE, ASIZE)
```

inquire the actual maximum display surface size of the workstation whose identifier is `WSID` These inquiries allow the sample program to run on all supported workstations.

Workstation State List

All inquiries whose first parameter is a workstation identifier, inquire information from the WSL corresponding to that open workstation. The values in the WSL reflect the current state of the workstation when the workstation is open.

For example, the Inquire Highlighting Filter **(GPQHLF)** subroutine will return the current highlighting filter lists for a specified workstation identifier.

Structure Related Inquiries

Structure related inquiries let your application inquire the existence and content of structures.

To inquire the contents of an element within a structure, the structure must be open. Your application must first set the current element pointer at the target element, then issue the appropriate inquiry.

The graPHIGS API provides a rich variety of inquire element subroutine calls. For example, to inquire the length (in bytes) and element code for a set of structure elements, use the Inquire List of Element Headers **(GPQEHD)** subroutine.

Next, to obtain the contents of a set of structure elements, use the Inquire List of Element Data **(GPQED)** subroutine. Both **GPQEHD** and **GPQED** return structure element information starting at the structure element pointed to by the current element pointer. The format of a structure element is described in *The graPHIGS Programming Interface: Technical Reference*.

Part 2. Advanced

Chapter 10. Advanced Concepts

This chapter introduces advanced concepts that are part of the graPHIGS API Version 2. These topics are based on proposed extensions to the PHIGS standard called ″PHIGS+″ concepts presented in this part of the book involve an organizational change that divides the graPHIGS API into two distinct parts; the graPHIGS API *shell* and the graPHIGS API *nucleus*. The shell is tightly coupled to the user application and performs syntax checking and building of the graphics data stream. The nucleus, on the other hand, is tightly coupled to the resources used by a graPHIGS API application such as structure stores and workstations. A nucleus manages the resources that may be shared simultaneously by a number of application processes. Each application process is a separate thread of execution with its own shell but attached to the same nucleus. By dividing your application into multiple processes, you can distribute parts of the application to the workstation as a Distributed Application Process (DAP).

Applications written to run under Version 1 of the graPHIGS API should still run under Version 2. For a further discussion of Version 1 and Version 2 compatibility, please see Appendix B. Compatibility.

Some of the advanced graPHIGS API capabilities include:

- Image display functions
- Advanced event handling; including update complete events and application defined event handlers
- Non-Uniform Rational B-Spline (NURBS) for curve and surface primitives
- Advanced rendering for complex designs
- Control over frame buffer operations
- Advanced input device operations and control
- Advanced structure editing functions
- Additional methods for controlling structure traversal
- Functions to set the line style pattern, marker shape, and hatch
- Advanced text control including filled characters, proportionally spaced characters, and shearing of characters

This chapter describes the basic concepts for controlling the various new components of the graPHIGS API products. These components are supported to satisfy the following requirements:

- Avoid duplicate graphical data on a host and a workstation
- Perform part of a host application within a workstation
- Provide multi-application access to a workstation

The following section explains how to create and manipulate graphical data on a remote workstation. It will also point out the best mechanism to support such remote data manipulation. Access to the same workstation from multiple application processes is described as well, and you will learn how to maintain the integrity and security of each resource and application process.

Following is a list of the major differences between Version 1 and Version 2 of the graPHIGS API:

- The graPHIGS API Version 2 is divided into two major components, a graPHIGS API shell and a graPHIGS API nucleus. An application process has only one shell, but it may be connected to one or more nuclei.
- The concept of central structure store is extended to multiple structure stores in a graPHIGS API nucleus. Structure stores are completely independent of each other. For example, an Execute Structure element within a structure store can only reference a structure within the same structure store. Each structure store can be shared by multiple application processes through multiple shells, and can be displayed on any workstation within the same nucleus.

- A workstation is not considered to be a resource directly owned by an application process but is considered to be a resource owned by a graPHIGS API nucleus which may be accessed by multiple application processes.
- All releases of the graPHIGS API on all levels of the operating system on the RS/6000 follow the Version 2 architecture.

The graPHIGS API Environment

The following is a typical example of the graPHIGS API environment:

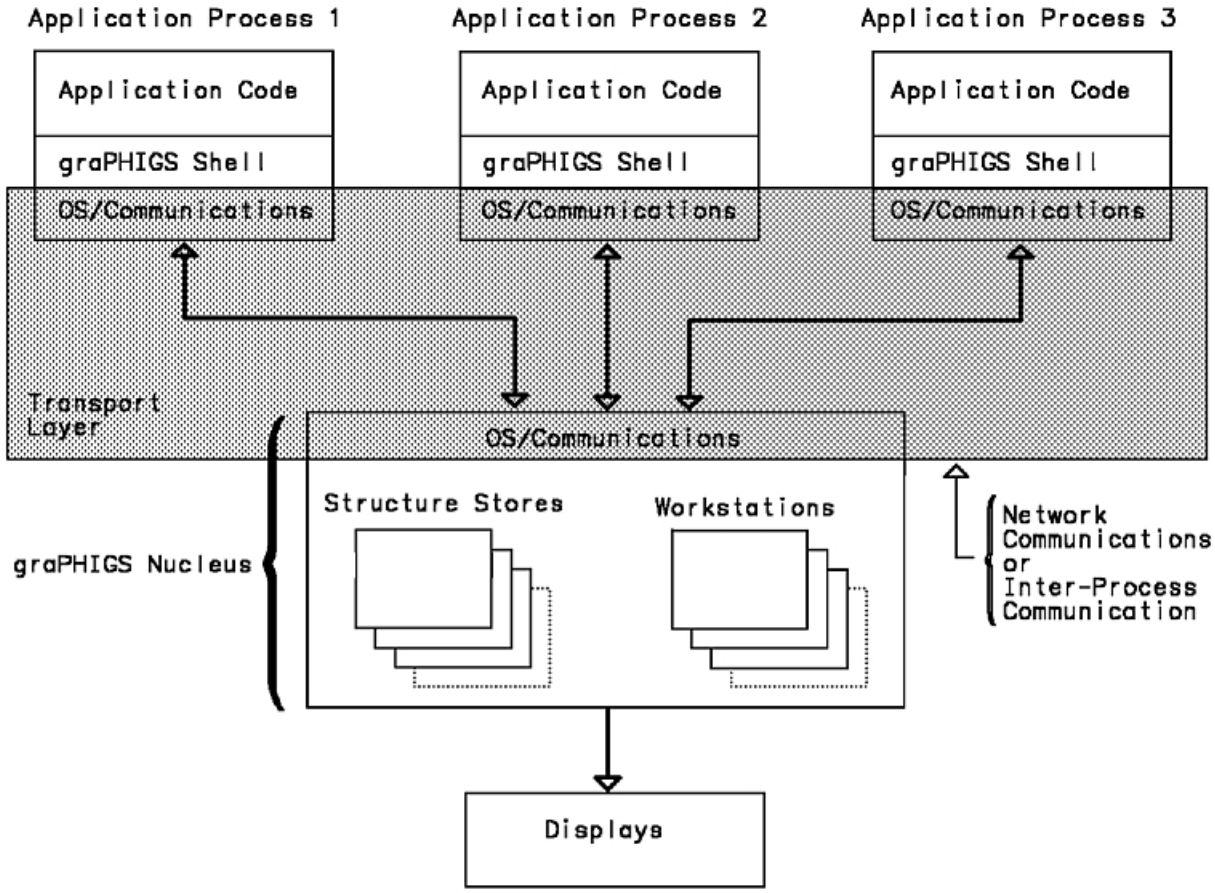


Figure 42. The graPHIGS API Environment. This illustration shows three application processes and their interaction with the graPHIGS nucleus. Each application process contains the application code, the graPHIGS shell, and OS/Communications. The graPHIGS nucleus consists of OS/Communications, structure stores, and workstations. The graPHIGS Nucleus directly interfaces with the display unit. The transport layer is formed by the OS/Communications portion of each application process and the graPHIGS nucleus. Network communication or inter-process communication takes place within this layer.

Components of the graPHIGS API Environment

Within this picture, the boxes roughly represent the following components:

Node A system consisting of a set of hardware and an operating system on which one or more tasks can run simultaneously.

Transport Layer
The transport layer provides communication between the shell and nucleus whether it is network

communication or inter-process communication. The communication methods used by the transport layer include SOCKETS, GAM, and direct subroutine calls.

Application
A unit of software that can be individually executed on a node. Each unit may be an entire application or may be one of several cooperative processes that comprise the application.

The graPHIGS API Shell
A component of the graPHIGS API products that controls communication between an application process and one or more nuclei.

The graPHIGS API Nucleus
The main component of the graPHIGS API products that performs most of the graPHIGS API functions. One graPHIGS API nucleus can handle multiple workstations, structure stores, image boards, and font directories. A nucleus can accept requests from multiple application processes through multiple graPHIGS API shells.

The graPHIGS API Workstation
A unit consisting of one logical display surface and a set of logical input devices which may correspond to a physical terminal, a virtual terminal or a window on a physical/virtual terminal.

Structure Store
A storage unit corresponding to the central structure store in the Version 1 graPHIGS API products. There may be multiple structure stores in a graPHIGS API nucleus. Structures in a structure store within a given nucleus can be displayed on any of the graPHIGS API workstations that are controlled by the same nucleus.

Font Directory
A font directory is a repository for application fonts that reside in the memory accessible to a nucleus.

Image Board
An image board is a nucleus resource that allows your application to store and display image data.

Resources of the Nucleus

As described above, structure stores, workstations, image boards, and font directories reside in a graPHIGS API nucleus and can be shared by multiple application processes, through multiple shells. Therefore, they are called *resources* of the graPHIGS API nucleus.

Within a nucleus, each resource is identified and referenced by a tag called a *resource identifier* A resource identifier is not the same as the identifier used in the graPHIGS API subroutine calls, such as the workstation identifier of the Open Workstation **(GPOPWS)** subroutine. To be able to use any of the advanced graPHIGS API functions, the workstation identifier is assigned by the application while the resource identifier is assigned to a resource by the nucleus when the resource is created. Regardless of how many application processes connect to the nucleus or what identifiers the processes use for the resource, each resource identifier is always unique within a specific nucleus. Nucleus resources which an application process can access are considered attached to the application.

When a resource is created at the request of your application process, it is implicitly attached to your application. A resource can also be attached to the application explicitly via the Attach Resource **(GPATR)** subroutine. In either case, your application process can assign its own identifier to the resource. Once a resource is attached to your application and has been assigned an application identifier, it is always referred to by the application identifier. Mapping between the application identifier and a resource identifier on the nucleus is automatically performed by the graPHIGS API shell.

The graPHIGS API Shell

An application process must have a graPHIGS API shell. Issuing the Open graPHIGS **(GPOPPH)** subroutine creates a graPHIGS API shell.

The graPHIGS API shell is responsible for receiving the parameters of the application's subroutine calls, validating them for correct syntax and for ensuring that calls are made in the correct sequence. The shell also builds the request in a form that is expected by the nucleus. This may involve data conversion such as IBM floating-point format to IEEE floating-point formats, ASCII-to-EBCDIC, and byte-swapping. The shell is also responsible for receiving responses and events from a nucleus and performing any necessary conversions. A *nucleus response* is the return of information to the shell that was directly requested by the shell through a previous request to the nucleus. Events are generated independently by the nucleus for a variety of reasons and sent to the shell asynchronously. Typical types of events include input device data and errors. The shell also provides trace facilities and error logging.

To perform these functions, each shell maintains the following information:

- For each connected nucleus:
  - A nucleus identifier
  - A communication path to the nucleus.
- For each resource which is currently attached to the shell:
  - An application identifier of the resource
  - A nucleus identifier which owns the resource
  - A nucleus identifier of the resource

Note that each resource has two identifiers, an identifier assigned by the application and a resource identifier assigned by a nucleus. By keeping this information, the shell can always perform mapping from an application identifier to a resource identifier assigned by a nucleus and vice versa. For almost all graPHIGS API subroutines, a resource (workstation, structure store, etc.) is referred to by its application identifier. However, when your application attaches to a resource that currently exists in a nucleus, it must use the resource's nucleus resource identifier. Use the Inquire Nucleus Resource Identifier **(GPQNCR)** subroutine to find the resource identifier of a resource assigned by the nucleus.

The graPHIGS API shell is terminated by the Close graPHIGS **(GPCLPH)** subroutine. If there are any nuclei connected to the shell, the Close graPHIGS subroutine call removes all of these connections.

Conceptually, a shell can access any nucleus on any node if there is a physical communication path between them. However, a given shell may have limitations as to which nuclei can be accessed, and which graPHIGS API subroutines are supported. Within each shell there is a graPHIGS API shell description table (PDT) and a graPHIGS API shell state list (PSL). By issuing the supported PDT and PSL inquiries, you may determine the capabilities, limitations, and state of a shell.

The graPHIGS API Nucleus

Generally speaking, a graPHIGS API nucleus exists and works independently of any graPHIGS API shell or application. A nucleus may run in a S/370, AIX, or 6090 environment and is created either through an environment-dependent mechanism outside of the graPHIGS API or when an application process tries to connect its shell to a nucleus.

A nucleus that exists in the same process is called a private nucleus. Only the shell that initiated the private nucleus may communicate with that nucleus. If a nucleus exists in a different process than the application, then that nucleus is called a remote nucleus (whether or not the nucleus resides on the same node as the application). A remote nucleus can communicate with any shell that is allowed to connect to it.

A special type of remote nucleus can exist on AIX: a child nucleus. A child nucleus is created when an application wants to connect to a private nucleus, but the nucleus has special requirements that it cannot fulfill inside the application's process (for example, not being threadable when running on a multiprocessing system or when the X Asynchronous Event Handling extension does not exist — for example, X11R6). Only the shell that parented the child nucleus may communicate with the child nucleus.

A connection between a graPHIGS API shell and a graPHIGS API nucleus is established by the Connect to Nucleus **(GPCNC)** subroutine with the following parameters:

- A nucleus identifier
- A nucleus connection method to be used in exchanging data with the nucleus
- A nucleus connection specification

The nucleus connection methods currently supported are listed below:

1=CALL
Use the call method when you want your application to use a private nucleus. For applications that run on S/390 or AIX, this is the default method (for applications that do not call the Connect to Nucleus subroutine).

2=GAM
Use the GAM method (Graphics Access Method) when your application is in a S/390 environment and the application wants to communicate with on of the following:

- 6090 nuclues
- remote nucleus on AIX using the graPHIGS API Gateway daemon
- remote nucleus on AIX using the 6098 with FDDI feature

3=SOCKETS
Use a TCP/IP sockets connection to communicate between an application running in a S/390 or an AIX environment and a remote nucleus on AIX. If both the application process and the nucleus are on the same node, then local sockets are used. If each process is on a different mode, remote sockets are used.

The figures below describe the connection methods and corresponding connection specifications supported by the different combinations of shell/nucleus environments:

Supported Connection Methods

Nucleus Environment \ Shell Environment	VM	MVS	AIX	6090
VM	CALL	N/A	N/A	N/A
MVS	N/A	CALL	N/A	N/A
AIX	GAM or SOCKETS	GAM or SOCKETS	CALL or SOCKETS	N/A
6090	GAM	GAM	N/A	SOCKETS

Connection Specifications

Application Environment \ Connection Method	1=CALL	2=GAM	3=SOCKETS
VM	Null (length=0)	Filedef (length<=8)	Nucspecx (length>=2)
MVS	Null (length=0)	DDNAME (length<=8)	Nucspecx (length>=2)
AIX	Null (length=0)	N/A	Nucspecx (length>=2)

```
          |      N/A      |      N/A      |      :0       |
     6090 |               |               | (length=2)    |
          ----------------------------------------------

N/A  Not a valid combination
x    Nucspec is set to the following specification:
        hostname: nucleus connection number

     Where:
      - hostname specifies the node id of an AIX
         system.
      - nucleus connection number specifies a
         a nucleus on the named system and is a number
         in the range of 0-255.
```

You may choose to have your application call the Connect to Nucleus **(GPCNC)** subroutine or have the graPHIGS API make the connect as part of the defaults processing performed by the Open graPHIGS **(GPOPPH)** subroutine. This may be done as follows if:

- Your application will be using a workstation resource supported in Version 1 and you want the graPHIGS API to perform the *nucleus connection processing*, then you do not need to do anything additional. When you execute your application, the graPHIGS API automatically connects to a nucleus with identifier=1 using the `CALL` connection method.
- Your application will be using a 6090 workstation resource, and you want the graPHIGS API to perform nucleus connection processing, then you must specify the default option (DEFNUC) in the External Defaults file (EDF) or Application Defaults Interface Block (ADIB) to indicate this.
- You want to explicitly connect your application to a nucleus, then you must specify the default option (DEFNUC) in the EDF or ADIB to prevent the graPHIGS API from doing the nucleus connection processing. See the *The graPHIGS Programming Interface: Technical Reference* for the correct syntax for specifying the DEFNUC default.

The most important role of the graPHIGS API nucleus is to distribute requests from graPHIGS API shells to nucleus resource handlers, and to distribute events or errors generated by nucleus resource handlers to graPHIGS API shells. For this purpose, the nucleus maintains the following information for each application process (shell) connected to the nucleus and each resource existing in the nucleus:

- For each application process (shell) connected to the nucleus:
  - A shell identifier
  - A message password
- For each resource existing in the nucleus:
  - A resource identifier
  - An identifier of the shell that created the resource
  - A resource password
  - A list of shell identifiers to which the resource is currently attached

Similar to the resource identifier, the identifier of each shell is unique within the nucleus and assigned by the nucleus when the shell is connected to the nucleus. This shell identifier is mainly used to validate an application's access to a resource. The application does not need to know the shell identifier except when the application wants to communicate with other applications through the graPHIGS API application message facility. To determine your shell identifier assigned by the nucleus, use the Inquire Shell Identifier **(GPQSH)** subroutine.

The connection between a shell and nucleus is terminated explicitly by the Disconnect Nucleus **(GPDNC)** subroutine and implicitly by the Close graPHIGS **(GPCLPH)** subroutine. When the connection is destroyed, all resources in the nucleus that are attached to the shell are automatically detached from it.

As with shells, some nuclei may have limitations such as the number or type of shells to which the nucleus can be connected, the number and type of resources that can be created, and the set of functions actually supported.

Within each nucleus there is a Nucleus Description Table (NDT) and a Nucleus State List (NSL). By issuing the supported NDT and NSL inquiries, your application may determine the capabilities, limitations, and state of a nucleus.

The following two sections describe workstation and structure store resources.

Structure Store

A structure store is a repository for structures. Once a connection between your shell and nucleus is established, your application can create a number of structure stores in the nucleus, limited only by the amount of memory available to the nucleus and its maximum supported number. The graPHIGS API shell associates one or more structure stores within the same nucleus to a workstation in order to produce output on the workstation. However, only one structure store may be associated with a workstation at any one time. Associating a structure store to a workstation implicitly disassociates a structure store previously associated. Several workstations within a nucleus may be associated to the same structure store. Your application program is responsible for managing structure names among the structure stores that it owns.

The Create Structure Store **(GPCRSS)** subroutine creates a structure store resource in the specified nucleus, and attaches it to the graPHIGS API shell. All structure stores are mutually independent. For example, a structure within a structure store can reference only structures within that structure store. Subroutines applied to a structure store affect only that structure store and have no side effects on other structure stores. The same structure identifier can be used in more than one structure store without conflict.

Your application can access multiple structure stores, therefore, it is important to select the desired structure store before editing. Selecting a structure store may be done implicitly when the graPHIGS API does nucleus connection processing or explicitly by your application calling the Select Structure Store **(GPSSS)** subroutine.

Structure stores can be created and manipulated using one of two methods. The default method, which provides compatibility between Version 1 and Version 2 of the graPHIGS API, relies on the automatic connection of a nucleus and the creation of a structure store. The structure store is also automatically selected to be the current structure store so that the following Version 1 code will continue to work:

```
Open graPHIGS (GPOPPH)
Open Structure (GPOPST)

.
.
.
Edit

.
.
.
Close Structure (GPCLST)
```

To suppress the default creation of both a nucleus and structure store or to allow an existing application to run on a 6090, your application must specify either an Application Defaults Interface Block (ADIB) or use an External Defaults File (EDF). ADIBs and EDFs are discussed in detail in *The graPHIGS Programming Interface: Technical Reference* When default nucleus and structure store processing creation is suppressed, your application will take the following form:

```
Open graPHIGS (GPOPPH)
   (specifying DEFNUC in ADIB or EDF)
Connect Nucleus (GPCNC)
Create Workstation (GPCRWS)
```

```
Create Structure Store (GPCRSS) or Attach Structure Store (GPATR)
Associate Structure Store to Workstation (GPASSW)
Select Structure Store (GPSSS)
Open Structure (GPOPST)

.
.
.
 (Edit Structure)

.
.
.
Close Structure (GPCLST)
Close Workstation (GPCLWS)
Close graPHIGS (GPCLPH)
```

In order to display the contents of a structure store on a workstation, the structure store must be associated to a workstation using the Associate Structure Store to Workstation **(GPASSW)** subroutine.

A structure store is automatically associated with a workstation when your application issues the Open Workstation **(GPOPWS)** subroutine, and a structure store is currently selected.

If your application has not selected the structure store or uses **GPCRWS** to create a workstation, then the structure store will not be automatically associated with the workstation. Using the Detach Resource **(GPDTR)** enables your application to detach a structure store from its shell. If **GPDTR** removes the final reference to the structure store, either through an attachment to a shell or association to a workstation, the structure store itself is released.

For each structure store within the nucleus, there is a Structure Store State List (SSL). By issuing the supported SSL inquiries, your application may determine the state of a structure store.

Workstations

The Create Workstation **(GPCRWS)** subroutine creates a workstation resource owned by a graPHIGS API nucleus and attaches the workstation to the graPHIGS API shell. **GPCRWS** requires the following parameters:

- A workstation identifier
- A nucleus identifier
- Length of the connection identifier
- A connection identifier
- A workstation type
- Zero or more PROCOPT descriptions

The **GPCRWS** subroutine replaces the Open Workstation **(GPOPWS)** subroutine and enables your application to specify on which nucleus the workstation is to be opened. It also enables your application to specify options for the workstation that were previously specified through the External Default File (EDF) or Application Default Interface Block (ADIB).

Using the **GPOPWS** subroutine creates a workstation resource on a nucleus with identifier 1.

A workstation can display graphical data from any structure store existing in the nucleus where the workstation resides. In order to specify which structure store is used for display, use the Associate Structure Store with Workstation **(GPASSW)** subroutine. A workstation can display graphical data within one structure store only. All display subroutines, such as Associate Root with View **(GPARV)** subroutine, are treated as referring to a structure within the structure store associated with a particular workstation.

Your application can detach a workstation from the graPHIGS API shell using the Detach Resource **(GPDTR)** or Close Workstation **(GPCLWS)** subroutine. **GPCLWS** detaches the workstation resource on a nucleus with nucleus identifier 1. If this subroutine removes the final reference to the workstation, the workstation and all its resources are removed from the nucleus.

For each workstation within the nucleus, there are two Workstation Description Tables (WDT) and a Workstation State List (WSL). By issuing the supported WDT and WSL inquiries, you may determine the capabilities, limitations, and state of a workstation.

The two WDTs for each workstation are:

- Generic Workstation Description Table - defines the maximum capabilities of the workstation.
- Actual (or Realized) Workstation Description Table - defines the actual capabilities of the workstation. This description table exists only after the workstation is opened.

To obtain the most accurate information about a workstation, you should inquire its capabilities from the actual workstation description table. In order to do this you must specify the actual workstation type on the inquiry. The actual workstation type is returned by the Inquire Realized Connection and Type **(GPQRCT)** subroutine.

Communication between a Shell and Nucleus

Typically, you won't need to know the details of the communication mechanism used between a graPHIGS API shell and nucleus. However, to achieve the maximum performance and to fully understand topics discussed in subsequent sections, you must have a general understanding of the communication mechanism.

Data Buffering

Data transmitted between a shell and a nucleus consists of a sequence of buffered data structures called *procedures* Most procedures correspond to a subroutine call in the graPHIGS API and are used by the graPHIGS API to pass information between a shell and nucleus. The length of a procedure is limited by the I/O buffer size that has a maximum value of 64K bytes. The most important consequence of this limitation is that the definition data of each output primitive cannot exceed 64K bytes. For example, the number of vertices specified in a Polyline 3 primitive must be less than approximately 5,000.

A graPHIGS API shell normally allocates two 64K I/O buffers, one outbound and one inbound, for each nucleus connection. However, when you want to minimize the amount of storage used by the shell, your application can override the default I/O buffer sizes through the COMBSZ default of an Application Defaults Interface Block (ADIB) or an External Defaults File (EDF).

By default, the data created by a shell is sent to a nucleus when one of the following situations occur:

1. When a graPHIGS API subroutine call requires modification of the content on the display surface of a workstation. The following subroutines are included in this category:
   - Set Deferral State **(GPDF)**
   - Update Workstation **(GPUPWS)**
   - Update Workstation Asynchronous **(GPUPWA)**
   - Redraw All Structures **(GPRAST)**
2. When a graPHIGS API subroutine call requires data to be retrieved from the nucleus. The data may be required for the shell but not for the application itself. The following subroutines belong to this group:
   - Set Password **(GPPW)**
   - Set Message Password **(GPMSPW)**
   - Request input device subroutine calls
   - Sample input device subroutine calls
   - Await Event

- Create Workstation (or Open Workstation) **(GPCRWS)**
- Create Structure Store **(GPCRSS)**
- Create Image Board **(GPCRIB)**
- Create Font Directory **(GPCRFD)**
- Attach Resource **(GPATR)**
- Detach Resource **(GPDTR)**
- Activate Font **(GPACFO)**
- Initiate Application Process **(GPINAP)**
- Send Private or Broadcast Message **(GPSPMS**,**GPSBMS)**
- Synch Request **(GPSYNC)** state list
- Inquire Requests for information contained in the nucleus (i.e., structure store, workstation and nucleus state list information)

3. When the buffer becomes full.

Two mechanisms are supplied for the application to change the default action for some of these situations. One is the Set Shell Deferral State **(GPSHDF)** subroutine that enables the application to suppress automatic data sending caused by the subroutines within the first group of this list. This enables the application to realize an animation sequence without causing unnecessary I/O operations between the shell and nucleus. For example, a series of alternating invocations of Set View Matrix and Update Workstation can be accumulated into the I/O buffer and sent to a nucleus with a single I/O operation. The nucleus would then generate multiple frames on the display surface without I/O overhead between each frame. Another mechanism to change the default action is provided by the Synchronize **(GPSYNC)** subroutine that enables the application to force data transmission directly. When the application issues Synch, any buffered requests are sent to the nucleus immediately.

Data is sent from a nucleus to a shell in the following situations:

1. When a request from a shell requires a response
2. When a request from a shell cannot be processed for some reason (an error will be generated)
3. When another application process requests an application message to be sent to a target application (shell)
4. When a component of the nucleus encounters an asynchronous error related to a resource attached to the shell
5. When an input device which has been activated by the shell generates an input device event
6. When certain situations generate events (that is, storage threshold has been reached or synchronous workstation updates have completed)

Unlike the data sent from a shell to a nucleus, the data from a nucleus to a shell will conceptually be transmitted as it is created. When more than one request from a shell is sent in a single I/O operation and there is more than one response or error related to these requests, the latter may also be buffered to increase I/O performance. However, such buffering is not visible to the application.

Resource Access Serialization

A nucleus can receive requests from multiple shells at any time and is only required to guarantee the following:

- Requests from any given shell are processed sequentially
- An individual request is processed completely before any other request is processed

There is no assurance that requests coming from multiple shells are processed in the global order of their creation. Therefore, when multiple shells share the same nucleus resource, special care must be taken to prevent the shells from ending up in a deadlock situation where both shells are waiting for the other shell to free up a resource.

Workstation Serialization: When a workstation is shared by multiple shells, usage of the workstation's input devices is serialized as follows:

1. When an input device is activated by an application process the device is dedicated to that process until the device is deactivated by the process or by an operator's action. An input device dedicated to a process cannot be accessed by any other process.
2. When an input device is set to Event or Request mode, only the application process that activated the device is given the input data.

   Likewise, when an input device is set to Request mode, only the application process that activated the device is notified when a break action occurs.
3. When the device is set to Sample mode, any application process can sample the device.

Structure Store Serialization: When a structure store is shared by multiple application processes, access to the structure store is also treated in a special manner. When an application process calls the Open Structure **(GPOPST)** subroutine, the currently selected structure store is dedicated to that application process. All subsequent structure modification requests from other application processes are not processed and are blocked by the nucleus, until the 'owning' application process calls the Close Structure **(GPCLST)** subroutine.

Notice that the structure store state described here is different from the structure state of the graPHIGS API shell. The structure state of the shell is mainly used to check whether an application process issues graPHIGS API calls in the correct sequence or not. For example, any output primitive subroutines will only be processed if the structure is opened to make sure these primitives are correctly stored in a structure. The purpose of the structure store state in the nucleus is not for validity checking but for ensuring that structure editing operations are consistent between application processes. For example, the Delete Structure **(GPDLST)** subroutine deletes not only the specified structure but also all references to the structure. If your application issues this subroutine call while another application process has a structure open, this automatic deletion of structure references (Execute Structure elements) can cause many problems.

All requests to a structure store that are blocked by this mechanism are not discarded but processed at a later time. There is no assurance that these blocked requests are processed immediately after the current owner issues a Close Structure. If the previous owner issues another Open Structure request, the structure store may be dedicated to the same application process again. Switching between requests from different application processes is not guaranteed to be performed in any particular order.

Font Directory Serialization: When a font directory is shared by multiple application processes, access to the font directory is treated in a special manner. When an application process calls the Load Font **(GPLDFO)** subroutine to load a font into the directory, the specified font directory is dedicated to that application process. All subsequent requests from other processes to access the font directory are blocked by the nucleus until the the load of the font completes. When the load of the font completes, the font directory may then be accessed by other application processes.

Serialization of Image Boards: When an image board is shared by multiple application processes, access to the image board is also treated in a special manner. When an application process calls the Write Rectangle **(GPWRCT)** subroutine, requests from other processes to access the image board are blocked by the nucleus until the write to the image board completes. When the write to the image completes, the image board may be accessed by other application processes.

Distributed Application Processes (DAPs)

The graPHIGS API gives you the ability to divide your application into multiple Distributed Application Processes (DAPs). Each DAP is a separate executable graPHIGS API process with its own graPHIGS API shell. These DAPS are downloaded to the remote nucleus by a connected shell process, using the Initiate Application Process (GPINAP) subroutine or the Execute Application Process (GPEXAP) subroutine. A

DAP that already exists on the node containing the remote nucleus can be executed by using GPINAP or GPEXAP. The use of DAPS allows you to offload some functions that cannot be performed efficiently across a network.

A DAP can perform many diverse functions, including:

- using a valuator to change a view or modeling transformation matrix.
- communicating with a device such as a digitizer that is attached to the node containing the remote nucleus
- performing animation

On a 6090, DAPs are limited by the amount of memory available and the fact that disk files are not available. The memory in a 6090 is shared by other components, such as the graPHIGS API nucleus, operating system, and other DAPS. Therefore, only those functions that can be performed faster in the 6090 Graphics System by eliminating the communications overhead with the host, should be performed with DAPS that run on a 6090.

On an AIX system, a DAP has the same privileges and restrictions as any other application on the system, as shown in the following figure:

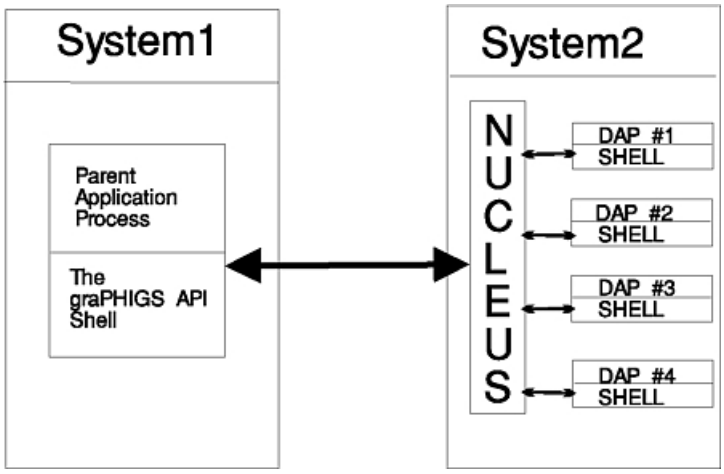


Figure 43. Dap Organization. This illustration shows how a DAP is like any other process on the system. In this illustration, system 1 is running a parent application process and the graPHIGS API shell. System two is running a graPHIGS nucleus and 4 DAPs. Each DAP runs its own shell and this shell interfaces with the nucleus. The nucleus in system 2 interfaces with the graPHIGS API shell and the parent application process.

For a discussion of compiling, starting, and terminating DAPs, see *The graPHIGS Programming Interface: Writing Applications*.

One method for two application processes to communicate is through the application message facilities provided by the graPHIGS API as discussed in the following section.

Communication between Multiple Application Processes

In order for multiple application processes to communicate, the graPHIGS API provides a message facility used to send application defined messages from one application process to the event queue of another process. The messages are sent using the Send Broadcast Message **(GPSBMS)** and Send Private Message **(GPSPMS)** subroutines and can be retrieved from the event queue by the target process using the Get Message **(GPGTMS)** subroutine.

The format of the message event is defined by the graPHIGS API, but there is no restriction on the contents of the application message. The graPHIGS API shell and nucleus do not perform any validity

checking or implicit conversion on the message data and no additional information, such as the originator of the message, is supplied in the message event. The graPHIGS APIthat performs the following conversions:

- EBCDIC to ASCII character encoding
- ASCII to EBCDIC character encoding
- IBM single precision floating point to IEEE single precision floating point
- IEEE single precision floating point to IBM single precision floating point
- Byte order swapping

GPCVD performs the conversions based on the environment your application is running in and the target environment specified as a parameter to **GPCVD** With **GPCVD**, one application process can convert data to a form recognized by the target environment or by its own environment. The data can be in the form of either a character string, integer array, floating-point array, or data record.

In order to prevent an application process from receiving unexpected messages sent by unknown applications, the following two levels of protection are provided:

1. When a shell is connected to a nucleus, the nucleus assigns the shell an identifier known only to the application which owns the shell. The identifier is called a **shell identifier** and can be retrieved through the Inquire Shell Identifier **(GPQSH)** subroutine.
2. Each application process can set a *message password* for its shell using the Set Message Password **(GPMSPW)** subroutine. As with passwords of resources, the message password can take one of the following values:

   0=NONE
   No application messages are received.

   -1=ALL
   All application messages are received.

   UNIQUE INTEGER
   Only the application message sent with the matching password are received.

The message password of a shell is set to `0=NONE` when a shell and nucleus are connected and can be set to another value using the **GPMSPW** subroutine.

The Send Private Message **(GPSPMS)** subroutine has the following parameters:

- A nucleus identifier to which the target shell is connected
- A shell identifier of the target shell
- A message password of the target shell
- A major code
- A minor code
- Length of the message
- A variable length byte string (the message)

This message is sent to the target shell only when the shell has message password ALL, or when the specified message password matches with that of the target shell.

Because the broadcast message has no specific target, the application need not specify any shell identifier or message password. The broadcast message is sent to all shells attached to the specified nucleus which have their shell password ALL. The Send Broadcast Message **(GPSBMS)** subroutine has the following parameters:

- A nucleus identifier
- A major code

- A minor code
- Length of the message
- A variable length byte string (the message)

A typical scenario for two applications processes to share a resource by using the application messages is presented below. The resource being shared is a structure store created by the first application process.

- ***Application Process 1:***
  1. Opens graPHIGS **(GPOPPH)**
  2. Connects to the nucleus **(GPCNC)**
  3. Obtains its own shell identifier from the nucleus **(GPQSH)**
  4. Creates a structure store resource **(GPCRSS)**
  5. Obtains the structure store resource identifier as it is known on the nucleus **(GPQNCR)**
  6. Assigns a password to the structure store resource **(GPPW)**
  7. Sets its own shell message password to ALL **(GPMSPW)** so it can receive broadcast messages
  8. Waits for broadcast messages from an application **(GPAWEV)**
- ***Application Process 2:***
  1. Opens graPHIGS **(GPOPPH)**
  2. Connects to the same nucleus **(GPCNC)** as application process 1
  3. Obtains its own shell identifier from the nucleus **(GPQSH)**
  4. Sets its own shell message password so it will receive messages only from processes that know the message password **(GPMSPW)**
  5. Sends a broadcast message to any application process connected to the same nucleus **(GPSBMS)**

     Note: The message contains the shell identifier and message password of application 2.
  6. Waits for a private message from an application **(GPAWEV)**
- ***Application Process 1:***
  1. Receives notification that an event has arrived **(GPAWEV)**
  2. Gets the data contained in the message event **(GPGTMS)**
  3. Uses **GPCVD** to convert the data to a form recognizable by the receiving environment.
  4. Sets its own shell message password to some unique value so it will not receive other broadcast messages **(GPMSPW)**
  5. Sends a private message to the application process 2 containing its own shell identifier, message password, the nucleus resource identifiers and passwords of the resources to be shared **(GPSPMS)**

     Note: A private message is used because application process 1 now has the shell identifier and message password of application process 2 from the data passed in the broadcast message.
- ***Application Process 2:***
  1. Receives notification that a message event has arrived **(GPAWEV)**
  2. Obtains data contained in the message event **(GPGTMS)**
  3. Uses **GPCVD** to convert the data to a form recognizable by the receiving environment.
  4. Attaches to the structure store resource created by application process 1 **(GPATR)**

Each process can now edit structures within the same structure store.

Resource Sharing

One application process can attach a nucleus resource (a workstation, structure store, image board, or font directory) created by another application process to its process by using the Attach Resource **(GPATR)** subroutine with the following parameters:

- Resource type

- An application identifier to be assigned to the resource
- A nucleus identifier which owns the resource
- A nucleus resource identifier of the resource
- A password assigned to the resource.

The resource type parameter must indicate one of the following resources:
- Workstation
- Structure store
- Image Board
- Font directory.

To be able to attach a resource, your application must know the resource's nucleus resource identifier and password. Initially, this information is known only to the application process that created the resource. The nucleus resource identifier is assigned by the nucleus when the resource is created and can be retrieved using the Inquire Nucleus Resource Identifier **(GPQNCR)** subroutine.

The password of each resource can have one of the following values:

0=NONE
Only the application that created the resource can access it

-1=ALL
Any application can access (attach) the resource

UNIQUE INTEGER
Any application with the correct password can access(attach) the resource.

Each time a resource is created in a nucleus, the resource's password is set to `0=NONE` so it cannot be shared by any other application process. Therefore, when two or more application processes want to share a resource, its password must be explicitly set to some value other than zero by the Set Password **(GPPW)** subroutine. For security reasons, this subroutine allows only the application process that created the resource to change its password.

A scenario for two application processes to share a resource follows:
- ***Application Process 1:***
  1. Opens graPHIGS **(GPOPPH)**
  2. Connects to the Nucleus **(GPCNC)**
  3. Creates a workstation resource **(GPCRWS)**
  4. Inquires the workstation resource identifier as it is known on the nucleus **(GPQNCR)**
  5. Assigns a password to the resource **(GPPW)**
  6. Passes the nucleus resource identifier and its password to application process 2.
- ***Application Process 2:***
  1. Opens graPHIGS **(GPOPPH)**
  2. Connects to the Nucleus **(GPCNC)**
  3. Attaches to the resource after receiving the resource identifier and password from Application 1 **(GPATR)**

The resource is being shared by the 2 application processes.

User Exits and Conferencing

The graPHIGS API supports user exits and provides subroutines that can work in conjunction with a conference utility. Although the software required in a conference utility is considerable in size, the benefits

can be great. Designers collaborating from remote sites can view the same model simultaneously, with each workstation able to interact with the application. New users of the graPHIGS API also benefit when getting assistance from the help desk through a conference utility. The following graPHIGS API subroutines are available for use with a conference utility.

- Inquire Nucleus Specification **(GPQNS)** returns the hostname for conference utility connection.
- Inquire Workstation Type and Options **(GPQWTO)** returns the workstation type and options for the master workstation where the application runs.
- Create Workstation **(GPCRWS)** opens participating workstations using the same type and options used for the master workstation.
- Inquire Input Device State **(GPQID)** returns the details of an input device's operating mode.
- Inquire device state subroutines return the state of input devices attached to participating workstations.
  - Inquire Choice Device State **(GPQCH)**
  - Inquire Locator Device State **(GPQLC)**
  - Inquire Pick Device State **(GPQPK)**
  - Inquire Stroke Device State **(GPQSK)**
  - Inquire String Device State **(GPQST)**
  - Inquire Valuator Device State **(GPQVL)**
- Set Input Device Mode **(GPIDMO)** sets the input device mode in order to request input from a specific device without locking out another participating workstation from input focus.

Refer to *The graPHIGS Programming Interface: Technical Reference* for more information about user exits and a description of a typical conference utility.

Chapter 11. Structure Elements

While preserving the hierarchical modelling capabilities found in PHIGS, the PHIGS PLUS standard provides a number of new functions which enable your application program to take advantage of the advanced features found in modern graphics workstations. These functions include support for:

- Lighting and shading (local lighting)
- Hidden line/hidden surface removal (HLHSR)
- Transparency
- Depth cueing
- Advanced primitives such as:
  - Triangle strip
  - Line and marker grids
  - Polyhedron edge
  - Parametric curves and surfaces
- Direct color

This chapter will help you understand the advanced graPHIGS API structure elements many of which are based on basic structure elements presented in the first part of this manual. Included is a discussion of advanced graPHIGS API primitives and attributes.

If you are already familiar with the graPHIGS API, you may wish to skip the section discussing structure element classifications. If you have a basic understanding of the graPHIGS API but would like a review of the basic structure elements, then read all of this chapter.

Structure Element Classifications

The graPHIGS API structure elements are grouped into the following classes:

- Output Primitive Elements
- Primitive Attribute Elements
- Transformation Elements
- Structure Execution Elements
- Generalized Drawing Primitives (GDPs)
- Generalized Structure Elements (GSEs)
- Application Data Elements

Each structure element class is defined briefly below:

Output Primitive Elements

Structure elements which generate visible output on a workstation are classified as output primitive structure elements. Polylines, polygons, surfaces, and curves are examples of output primitive elements supported by the graPHIGS API A large portion of this chapter is devoted to explaining advanced primitives such as curves and surfaces.

Primitive Attribute Elements

The appearance of an output primitive is controlled by attributes applied to the primitive when the primitive is drawn. Primitive attribute structure elements define attributes such as color, line type, pick identifiers, and rendering quality. Default values are used for attributes which are not specified by your application. Some of the advanced output primitives discussed in this chapter also contain attribute data as part of their definition.

Attributes are specified either individually or through an index into an attribute bundle table. For more information on bundled attributes, see Chapter 4. Structure Elements in Part 1 of this manual.

Transformation Elements

Transformation elements are used to position and orient output primitive elements in 2D and 3D space. A transformation element consists of a 2D or 3D transformation matrix which is inserted into the open structure. All primitives following the transformation matrix are transformed to a new position and orientation in 2D or 3D space.

Structure Execution Elements

The graPHIGS API gives you the ability to organize your graphics data into hierarchical structure networks much like applications are organized into modules. For example, the geometry of a car can be represented by a network of structures. The first structure, called the root structure, would contain the geometry of the car body. The second structure, called a sub-structure, could contain the geometry of one wheel. Through the use of four transformation structure elements and four structure execution elements, the body structure can position and execute the wheel structure to draw a car with four wheels. Two extensions to the concept of structure execution are conditional structure execution and conditional structure return. Both extensions are discussed in detail in Chapter 12. Structure Concepts.

Generalized Drawing Primitives

Generalized Drawing Primitives (GDPs) are structure elements optionally supported on specific hardware. For example, the circle primitive is not supported on 5080 Model 1 devices but is supported on 5080 Model 2 devices. Therefore, the circle primitive is defined as a GDP. The list of GDPs supported on a specified device can be inquired using the Inquire List of Generalized Drawing Primitives **(GPQGD)** subroutine. The list of attributes used by a GDP can be inquired using the Inquire Generalized Drawing Primitive **(GPQGDP)** subroutine. Many of the advanced primitives discussed in this chapter are defined as GDPs. Therefore, before using a GDP primitive, your application should inquire whether or not the GDP is supported on the workstation you are using.

Generalized Structure Elements

Typically, Generalized Structure Elements (GSEs) represent attributes or control subroutines that are optionally supported on specific hardware. For example, the conditional execute structure element is defined as a GSE because it is not supported on all hardware platforms. The list of GSEs supported on a specified device can be inquired using the Inquire List of Available GSEs **(GPQGSE)** subroutine.

Application Data Elements

Your applications can use the graPHIGS API structure storage facilities to store application specific data inside a structure using the Insert Application Data **(GPINAD)** subroutine. The data is stored in the form of a data record and can be retrieved using the Inquire List of Element Headers **(GPQEHD)** and Inquire List of Element Data **(GPQED)** subroutines. Using the graPHIGS API structure store to store large amounts of data, data which will change frequently, or data which will be retrieved frequently is not recommended due to the overhead associated with storing and retrieving data to and from a structure store.

Output Primitive Elements

The next sections discuss the following output primitive structure elements:

- Output Primitive Classifications
- Reference Vectors
- Composite Primitives
- Geometric Normals
- Advanced Output Primitives

Basic structure elements are discussed in Chapter 4, Structure Elements in Part 1 of this book.

Output Primitive Classifications

The graPHIGS API primitives can be grouped into the following geometry classes:

Class	Example Primitives
Point	Polymarkers, Annotation Text
Curve	Polyline, Circle, Arc
Surface	Polygon

Each primitive class has a measure (for example, length, area) corresponding to its dimensionality. Points have no dimension and therefore have no measure. The measure of a curve is the length of the curve and is therefore one-dimensional. A surface has area as its measure and is therefore two-dimensional.

Geometrical entities other than points are conceptually infinite in their dimensions. With a primitive, however, bounds are placed on the geometry so that only a subset of the entity is displayed. For example, a line segment generated by the graPHIGS API polyline primitive is a subset of a line and is defined by two points. The two points define the infinite line as well as the limits of the line segment. We can generalize this concept to say that each graPHIGS primitive is based on a geometrical entity that is limited by entities within the next lower geometrical class. Another example of this concept is the polygon primitive which is comprised of a plane that has been limited by a set of line segments.

The PHIGS standard defines only a few primitives within each geometry class. Most other common primitives can be approximated using this basic set. For some primitives such as arcs and curves, the quality of the approximation deteriorates as the primitive is scaled due to modeling or viewing transformations. Another negative side effect of approximating all entities with a basic set is that complex entities require large amounts of space, thus impacting performance and memory utilization. The graPHIGS API includes a broader set of primitives to alleviate these problems. The supported primitives are summarized in the following table which is organized by geometry class and boundary specification type.

Class	No Limits Required
Point	Polymarker Pixel Write Annotation Text Annotation Text Relative Marker Grid

Class	Coordinate Space Limits	Parameter Space Limits
Curve	Polyline Disjoint Polyline Polyline Set with Data Geometric Text[default] Character Line Polyhedron Edge	Non-Uniform B-Spline Curve Circular Arc Elliptical Arc Line Grid
Surface	Polygon Polygon with Data Geometric Text[default] Triangular Strip Quadrilateral Mesh Composite Fill[default]	Non-Uniform B-Spline Surface Trimmed Non-Uniform B-Spline Surface

Notes:

1. When character definition indicates NO FILL
2. When character definition indicates FILL
3. The elements of this primitive include both coordinate space and parameter space limits.

Advanced Output Primitives

The graPHIGS API supports a wide variety of advanced output primitives. Many of the advanced primitives are not supported on some hardware platforms. Those primitives that are supported are listed in *The graPHIGS Programming Interface: Technical Reference* and can be inquired using the Inquire List of Generalized Drawing Primitives **(GPQGD)** subroutine. **GPQGD**'(*wstype*). To obtain the ″realized″ workstation type for a specific workstation, use the Inquire Realized Connection and Type **(GPQRCT)** subroutine. A realized workstation type is a unique workstation type generated when a workstation is opened and provides a means of inquiring the actual configuration of a workstation. In combination with **GPQRCT**, the **GPQGD** subroutine will return the actual output primitive capabilities of the workstation your application is using.

Reference Vectors

The definition of some advanced primitives require reference vectors to be specified. In order to minimize traversal time and data stream size, the graPHIGS API stores only the normalized form of these vectors. A normalized vector is defined as follows:

$$N=\left(\frac{vx}{L}, \frac{vy}{L}, \frac{vz}{L}\right)$$

where:

$$L = \sqrt{vx^2 + vy^2 + vz^2}$$

Therefore the length of the resulting vector (*N*) is 1.

When your application inquires the content of some primitive structure elements, the vector may not be exactly the same as when it was originally given to the graPHIGS API Note that there is no loss of information since the same visual result would be achieved if your application placed the vector returned by Inquire Element Data **(GPQED)** subroutine back into the structure element.

Composite Primitives

For the basic polygon primitive, the boundary for a filled polygon can only contain line segments. There is no way to define a boundary made up of circles, arcs, and parametric curves. Also, with the introduction of parametric surfaces, there is a requirement to define an arbitrary boundary in the parameter space of the surface that identifies the parameter range of the surface to be rendered. This type of primitive is termed a **trimmed surface**.

Therefore, to permit your application to define primitives with a mixture of boundary element types, the following two composite primitives have been defined for the graPHIGS API:

- Composite Fill Area
- Trimmed Non-Uniform B-Spline Surface

These primitives simplify the definition of complex objects and the processing of the primitives when they are drawn. Both primitives are discussed in detail later in this chapter.

Geometric Normals

Surface primitives such as polygons and NURBS surfaces have a front side and a back side defined by the geometric normal of the surface. The normal is used when processing the surface for such operations as shading and culling.

Given a surface normal, the API uses the following rule to determine which side of the surface is the front:

> If the tail of the geometric normal is placed on the surface, the front side is the one which is seen when sitting on the head of the normal looking toward the tail.

The following figure illustrates the above rule:

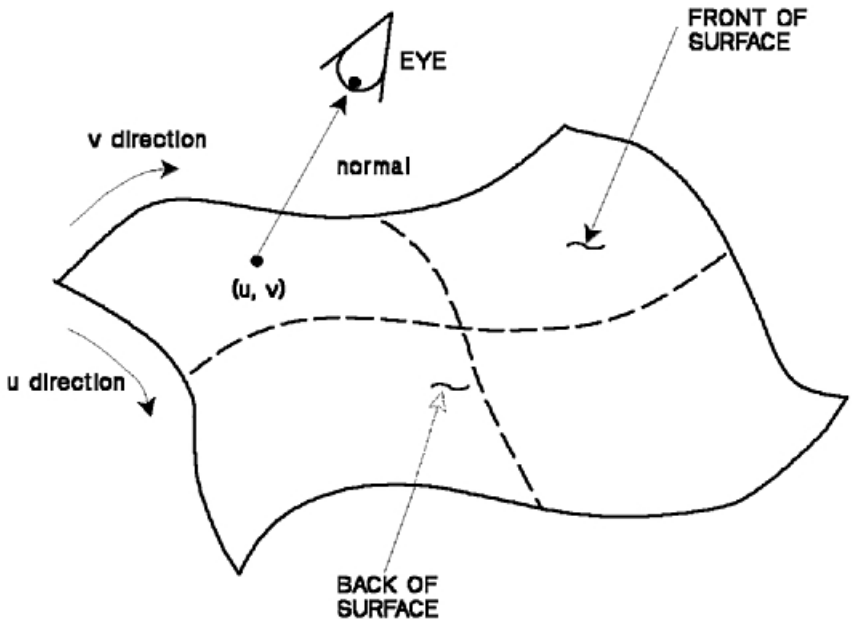


Figure 44. Determining the front and back faces of a surface. This illustration shows how graPHIGS uses geometric normals to determine the front and back faces of a surface. The front of the surface is the surface that allows the geometric normal to point towards your eye.

The geometric normal may be specified by your application for some primitives or will be calculated by the graPHIGS API The geometric normal is usually the normal to the surface but your application can specify it otherwise. For planar primitives, the geometric normal is used in place of vertex normals when the vertex normals are not specified by your application. The method used by the graPHIGS API to calculate the normal will be described for each surface primitive, so that you will be able to predict which side of a primitive will be the front.

Disjoint Polyline Primitive

When you need or want to store, as one element, many related but unconnected line segments, use the Disjoint Polyline **(GPDPL2** and **GPDPL3)** subroutines to create disjoint polyline structure elements. Disjoint polylines are similar to polylines in all respects except that they let your application store unconnected lines within a single primitive structure element.

Both **GPDPL2** and **GPDPL3** enable your application to draw a line segment, move to the next point without drawing, then draw a line segment within the same primitive. To accomplish this, the graPHIGS API accepts as input a move and draw array that specifies whether to draw a line segment from the current point to the next point or to move to the next point without drawing.

The following figure shows the arrays used to draw the pictured cube with a single disjoint polyline output primitive:

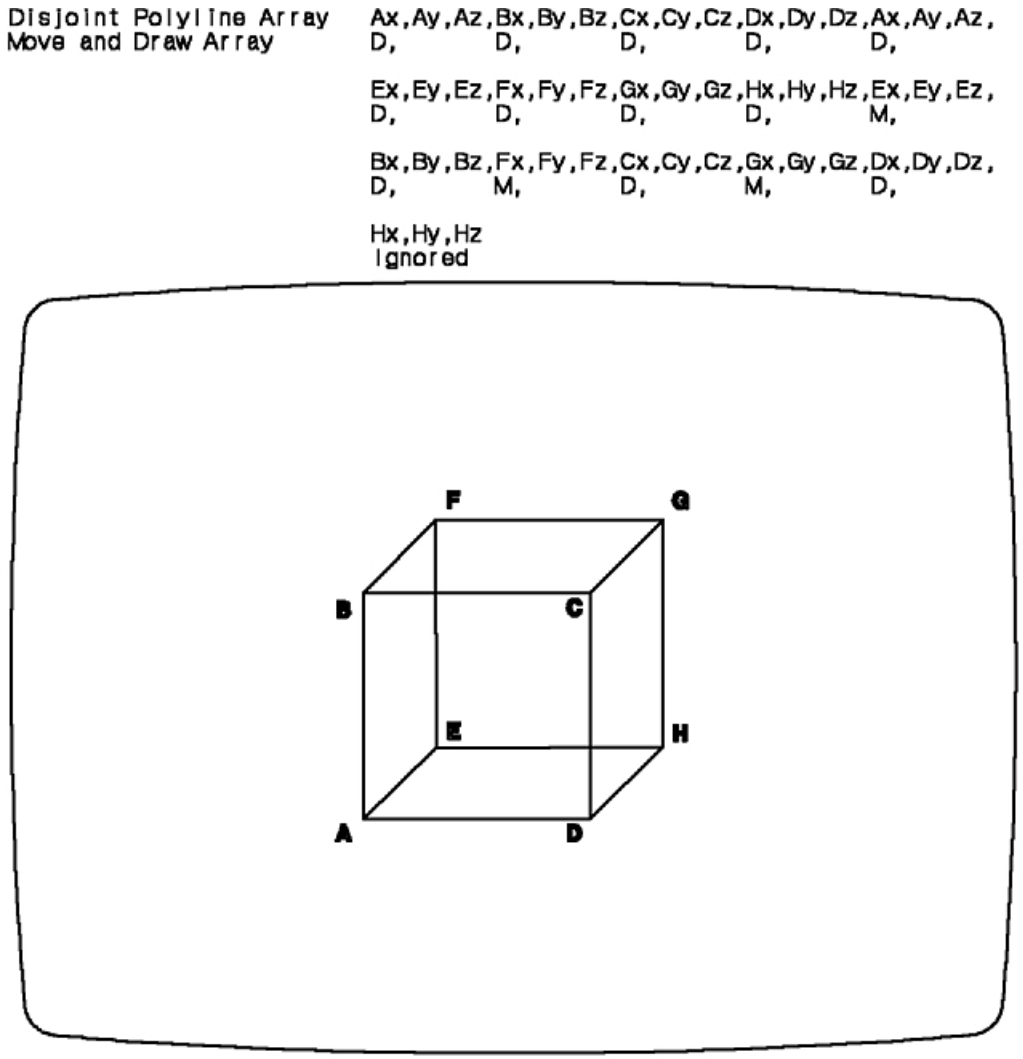


Figure 45. Disjoint Polyline Data and Cube. This illustration defines the arrays used to draw a cube with a single disjoint polyline output primitive. Two arrays are described: the disjoint polygon array, and the move and draw array. Vertices A, B, C, and D (clockwise from the lower left-hand corner) make up the front face of the cube; E, F, G and H (clockwise from the lower left-hand corner) create the back face of the cube. The disjoint polygon array is defined as follows: Ax, Ay, Az, Bx, By, Bz, Cx, Cy, Cz, Dx, Dy, Dz, Ax, Ay, Az, Ex, Ey, Ez, Fx, Fy, Fz, Gx, Gy, Gz, Hx, Hy, Hz, Ex, Ey, Ez, Bx, By, Bz, Fx, Fy, Fz, Cx, Cy, Cz, Gx, Gy, GZ, Dx, Dy, Dz, Hx, Hy, Hz. The move and draw array is defined as follows: D, D, D, D, D, D, D, D, D, M, D, M, D, M, D, ignored. The last value of ignored means that either a D or M may be specified but this value will be ignored since no vertex follows vertex H. The resulting three-dimensional cube is shown.

Polyline with Data

Like the Disjoint Polyline Primitive, Polyline Set 3 with Data **(GPPLD3)** allows you to store as one element, many related but unconnected line segments. During traversal, these elements generate a sequence of polylines from the specified list of points. Polyline Set 3 with Data also allows you to specify color per vertex.

You can control the color of a line segment three ways: 1) through use of the current polyline color, 2) through the color of the associated vertex, and 3) by interpolating between the colors of the vertices at the endpoints of the line.

The first method requires no color definition beyond that which you specify for the polylines. For the second and third methods, you must specify vertex colors with the primitive. The third method also requires you to insert a Set Polyline Shading Method attribute. See Set Polyline Shading Method for more information.

Set Polyline Shading Method

Use Set Polyline Shading Method **(GPPLSM)** when you want a gradual change in color (interpolation) across polyline segments spanning vertices of different colors. If you are specifying that polyline shading color be used with **GPPLSM**, you must be sure to define vertex colors in the Polyline Set 3 with Data primitive.

Pixel Primitive

With the graPHIGS API, your application can control the color of individual pixels on a workstation display through the Pixel **(GPPXL2** and **GPPXL3)** subroutines. Your application defines the color of each pixel as in index into the workstation's default color table. The default color table for most workstations is the display color table which implies that the content of this primitive is copied directly to the frame buffer. For more information on color tables, see Chapter 17. Manipulating Color and Frame Buffers.

In addition to the array of pixel values, your application specifies a point in Modeling Coordinates (MC) at which to position the array of colors. That specified point corresponds to the top left corner of the pixel rectangle.

The following figure shows how transformations affect only the location of the pixel data. Note that not all devices support clipping of the pixel primitive.

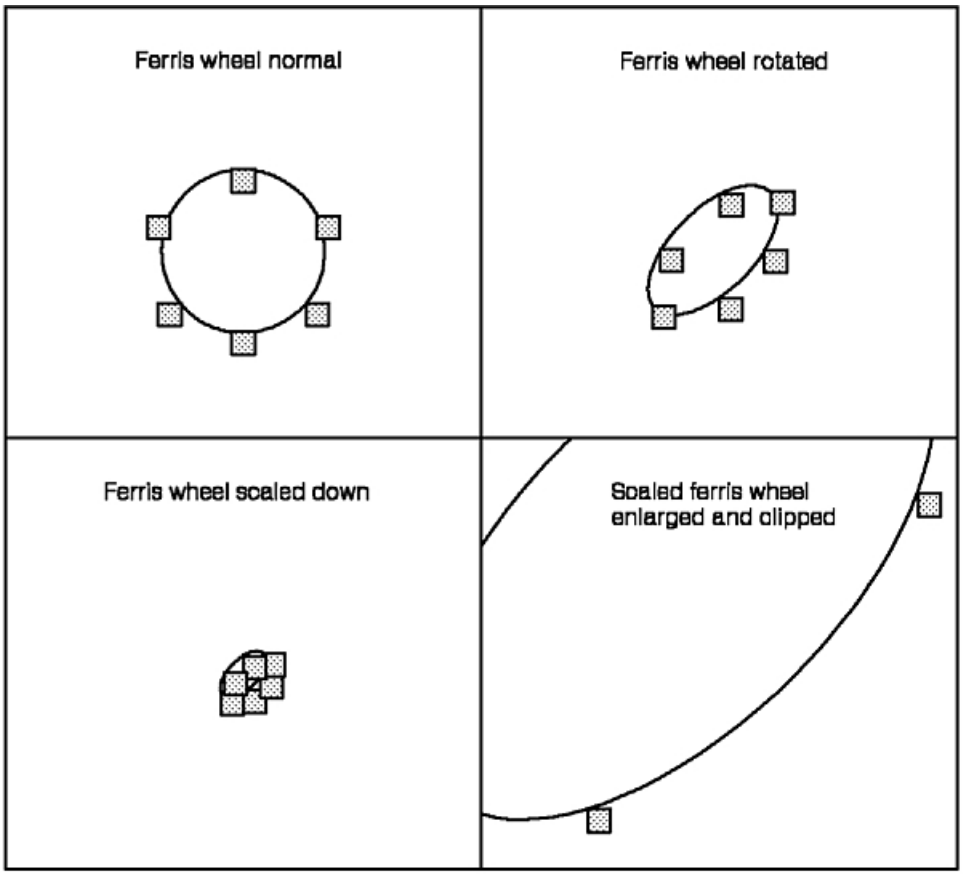


Figure 46. The Pixel Output Primitive. This illustration shows how transformations affect only the location of the pixel data. Four cases are shown in this illustration: normal, rotation, scaling down, and enlarging and clipping. In all four cases, none of the pixel data itself is modified, only the location of the pixel changes relative to the transformation being performed.

When a pixel primitive is encountered during structure traversal, each pixel index is written to the frame buffer in a workstation-dependent way depending upon the size of the workstation's frame buffer. For example, for an 8-bit frame buffer, the lower order 8-bits of each pixel index are written to the frame buffer. For a 24-bit frame buffer, the lower order 8-bits of each pixel value are duplicated in each component of the frame buffer. For more information on colors, frame buffers, and the color pipeline, see Chapter 17. Manipulating Color and Frame Buffers.

Circle and Circular Arc Primitive

The Circle 2 **(GPCR2)** subroutine lets your application create structure elements which define circles by specifying a circle's center point and a radius. The center point is specified in X and Y Modeling Coordinates and the Z coordinate is assumed to be zero. Your application can create a 3D circle using the ellipse subroutine. Circle primitives are drawn as polylines and are not filled.

The Circular Arc 2 **(GPCRA2)** subroutine lets your application creates a structure element which defines a circular arc (that is, a portion of a circle's perimeter). In addition to the arc's center point and radius, two angles are specified that define the portion of the circle's perimeter to be displayed. The arc is drawn in the counter-clockwise direction from the starting angle position to the ending angle position in the Z=0 plane.

Ellipse and Elliptical Arc Primitive

The ellipse structure element defines an ellipses in either 2D or 3D Modeling Coordinates (MC) and can be created using the Ellipse 2 **(GPEL2)** or Ellipse 3 **(GPEL3)** subroutines. Ellipses are drawn as polylines and are not filled.

An ellipse is defined by a center point and two vectors originating from the center point. The magnitude and direction of the first vector define the major axis of the ellipse. The second vector defines a point on the perimeter of the ellipse. For **GPEL2**, the ellipse is drawn through the two points in the Z=0 plane. For **GPEL3**, the ellipse is drawn through the two points in the plane specified by the two vectors.

The elliptical arc GDPs **(GPELA2** and **GPELA3)** let your application display a portion of the perimeter of an ellipse. The arc is defined by two parameter values that define the start and the end of the arc. The arc is drawn in the direction of increasing parameter values.

The following is the basic form of the parametric ellipse:

$\mathbf{P}(t) = \mathbf{A} \times \cos(t) + \mathbf{B} \times \sin(t) + \mathbf{C}$

where:

A is the vector defining a point on the ellipse relative to the center

B is the vector defining another point on the ellipse relative to the center

C is the vector defining the center of the ellipse

t is the parametric variable.

Note that **A** and **B** are not required to be orthogonal.

The drawing direction of an ellipse can be changed using one of the following methods:

- Interchanging vectors **A** and **B**
- Interchanging the parameter limits t_1 and t_2
- Adjusting the parameter limits by pi/2 (t_i = pi/2 - t_i).

The following table summarizes the parameters of this equation for each of the graPHIGS API primitives:

	Full (closed)	**Arc**
Ellipse	**C** = (C_x, C_y, C_z) **A** = (A_x, A_y, A_z) **B** = (B_x, B_y, B_z) t_1 = 0 t_2 = 2 pi	**C** = (C_x, C_y, C_z) **A** = (A_x, A_y, A_z) **B** = (B_x, B_y, B_z) t_1 = specified t_2 = specified
Ellipse 2	**C** = (C_x, C_y, 0) **A** = (A_x, A_y, 0) **B** = (B_x, B_y, 0) t_1 = 0 t_2 = 2 pi	**C** = (C_x, C_y, 0) **A** = (A_x, A_y, 0) **B** = (B_x, B_y, 0) t_1 = specified t_2 = specified
Circle 2	**C** = (C_x, C_y, 0) **A** = (r, 0, 0) **B** = (0, r, 0) t_1 = 0 t_2 = 2 pi	**C** = (C_x, C_y, 0) **A** = (r, 0, 0) **B** = (0, r, 0) t_1 = specified t_2 = specified

Polygon with Data Primitive

A polygon with data primitive is an extension of the polygon primitive that defines a polygon from a set of contours which can be rendered with one of several interior styles. In addition to the vertex coordinates and sub-area definitions, this primitive definition can include additional information to control the rendering process. For the two- and three-dimensional forms of this primitive, vertex colors, convexity flags, and boundary flags may be specified. The geometric normal of the polygon and/or a normal for each vertex

may be specified for the three-dimensional form only. The specified colors and normals are used in rendering the primitive. For example, the normals may be used in the lighting calculations to produce more realistic effects.

The convexity flag indicates that the application has determined the convexity of the polygon to be either CONCAVE or CONVEX. As a result, the graPHIGS API does not have to determine the convexity of each polygon every time the primitive is rendered, and rendering performance is improved. Sample C subroutines for determining the convexity of a set of polygons is provided in The graPHIGS Programming Interface; under the following directory:

```
/usr/lpp/graPHIGS/samples/convexcheck
```

Provided with the sample subroutines is a README file which explains their use.

A polygon with data primitive is created by calling the Polygon with Data **(GPPGD2** and **GPPGD3)** subroutines. The following figure shows a polygon with vertex normals and a geometric normal:

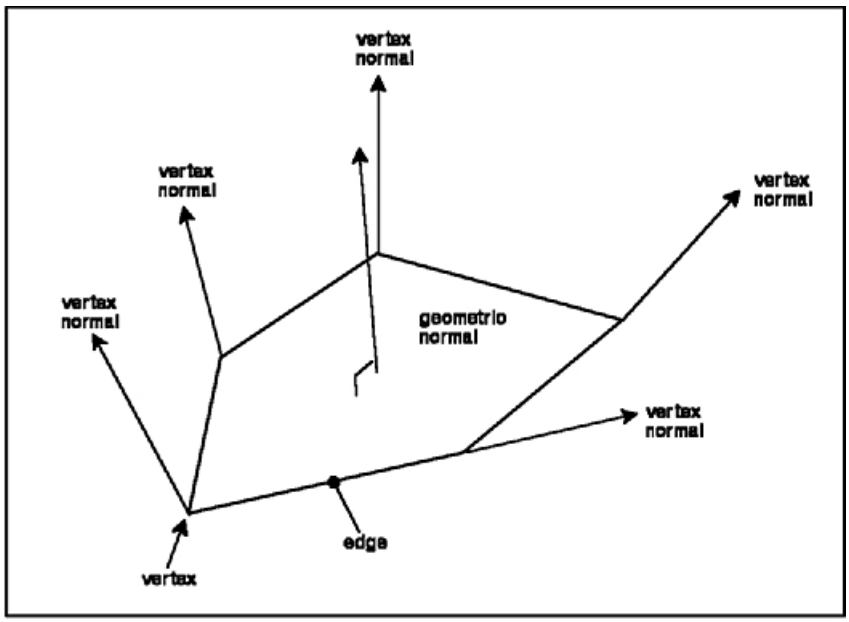


Figure 47. Five-sided polygon showing vertex normals and geometric normal. This illustration shows that each vertex in the polygon has a vertex normal specified. The polygon also has a geometric normal specified, and this normal is orthogonal to the surface of the polygon.

Note that the normals do not have to actually be orthogonal (normal) to the polygon.

The edges of this primitive are the same as for polygon except when the boundary flags are specified. In this case, only a subset of the polygon's perimeter will comprise the edge. The Edge Flag attribute still has the same meaning for this primitive as for polygon, which is to control whether the defined edges are rendered or not. Notice that when the Edge Flag attribute is set to `1=OFF`, the boundary flags have no effect. When it is set to `2=ON`, only those parts of the polygon's perimeter defined by the boundary flags will be rendered as part of the edge.

As for polygon 3, it is the responsibility of your application to ensure that all vertices of the three-dimensional form of this primitive are co-planar. The effect produced by non-planar polygons is undefined.

Triangle Strip Primitive

To create a triangle strip such as the one shown in the following figure, your application can call the Triangle Strip**(GPTS3)** subroutine. The triangle strip primitive generates a sequence of *n*-2 triangles from *n* vertices. The k^{th} triangle is defined by vertices *k*, *k*+1 and *k*+2

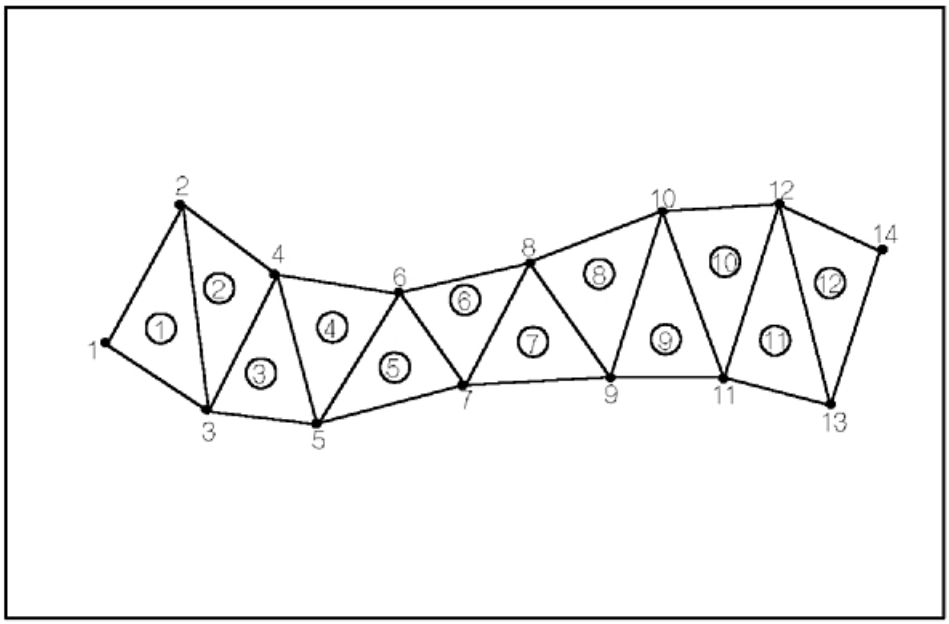


Figure 48. Triangle strip primitive. This illustration shows a series of triangles strung together to form a triangle strip. This triangle strip contains 14 vertices; therefore, the triangle strip is composed of 12 triangles. All the even vertices are on the top of the primitive, and all the odd vertices are on the bottom. The first triangle is formed by vertices 1, 2, and 3; the second is formed by 2, 3, and 4, and so on.

As with polygon with data primitive, additional information may be specified to control the rendering of the primitive more precisely. Colors and normals may be specified for each triangle vertex as well as a geometric normal for each triangle.

The edges of this primitive are the line segments forming the boundaries of all triangles in the strip.

Note: Unexpected results may be produced on some workstations when a line type other than solid is selected.

Quadrilateral Mesh Primitive

To create a quadrilateral mesh primitive such as the one shown in the following figure, use the Quadrilateral Mesh 3**(GPQM3)** subroutine.

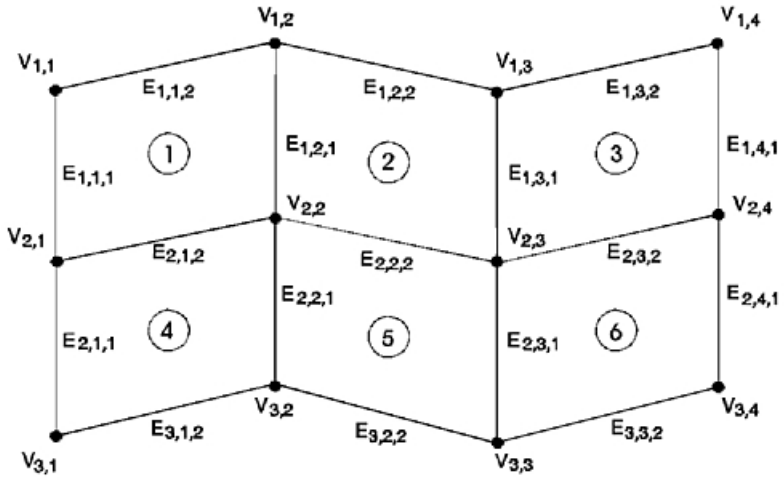


Figure 49. Quadrilateral Mesh Primitive. This illustration shows a quadrilateral mesh primitive made up of six quadrilaterals (two rows of three). The vertices are organized into 3 rows of 4 vertices. The top row contains vertices (listed from left to right) $V_{1,1}$, $V_{1,2}$, $V_{1,3}$, and $V_{1,4}$. The middle row contains $V_{2,1}$, $V_{2,2}$, $V_{2,3}$, and $V_{2,4}$. The last row contains $V_{3,1}$, $V_{3,2}$, $V_{3,3}$, and $V_{3,4}$. The first quadrilateral is created by connecting vertices $V_{1,1}$, $V_{2,1}$, $V_{2,2}$, and $V_{1,2}$; the second is created by connecting $V_{1,2}$, $V_{2,2}$, $V_{2,3}$, $V_{1,3}$; the third connects $V_{1,3}$, $V_{2,3}$, $V_{2,4}$ and $V_{1,4}$; the fourth connects $V_{2,1}$, $V_{3,1}$, $V_{3,2}$, and $V_{2,2}$, and so on.

Note: This figure shows numbering of vertices for a 4 by 3 primitive.

The quadrilateral mesh primitive generates a sequence of *m*-1 by *n*-1 quadrilateral facets from *m* by *n* vertices. Each set of four neighboring vertices defines a quadrilateral. Thus, each vertex you specify, except those at the ends, is used by four different facets. In this way, the application makes more efficient use of your vertices than it does when generating triangle strips.

You are not required to specify that the vertices of each quadrilateral lie in the same plane, however, the method of rendering such a quadrilateral depends on the workstation used.

If you specify fewer than two vertices in either direction, then the primitive is not visible.

You may specify additional data to control such aspects of rendering as color and edges.

Character Line Primitive

A character line primitive is created using the Character Line 2 **(GPCHL2)** subroutine. An integral number of a specific character is generated along a line between two end points by this primitive. It is classified as a text primitive and therefore uses most of the text attributes. The specification of this primitive includes the two end points, a nominal height in Modeling Coordinates and a single byte character code of the character to be used. Note that double byte characters cannot be used.

The current character set is bound to this primitive at definition time. The combination of text font, character set and character code at traversal time determine the actual set of strokes that are generated at each character position along the line. The actual height of the characters for a specific character line primitive will be the product of the nominal height from the primitive specification and a new attribute called character line scale factor. The character height attribute is ignored. As stated before, an integral number of characters is used to represent the line. As the character line primitive is scaled, the number of characters along the line will change but the height remains constant. The expansion factor is modified to ensure that an integral number of characters is produced and that there is no space at either end of the line segment. Text alignment and character spacing are not applied to this primitive. See Character Line Attributes for details.

Marker Grid Primitive

A marker grid primitive defines a planar grid of markers. The two-dimensional form of this primitive creates the grid in the XY plane (Z=0.0) of Modeling Coordinate space. The three-dimensional form specifies an arbitrary plane in three space in which the markers will be positioned. To create a two-dimensional or three-dimensional marker grid, your application should call the Marker Grid 2 **(GPMG2)** or Marker Grid 3 **(GPMG3)** subroutines respectively.

Each position on the grid is defined by the following parametric vector equation:

$$\mathbf{P}_{i,\,j} = \mathbf{P}_0 + i\mathbf{V}_1 + j\mathbf{V}_2$$

$$i_{min} <= i <= i_{max}$$

$$j_{min} <= j <= j_{max}$$

where:

+-

$\mathbf{P}_{i,\,j}$ is the position of the marker at grid location *i, j*

$\mathbf{P}_0$ is a 2D or 3D modeling coordinate which defines the origin of the grid

$\mathbf{V}_1$ is a 2D or 3D vector that defines the location of grid location *i*+1, *j* relative to grid location *i, j*

$\mathbf{V}_2$ is a 2D or 3D vector that defines the location of grid location *i*, *j*+1 relative to grid location *i, j*

i_{min} , i_{max} are the limits for parameter *i*. Can be any integer (positive, negative or zero).

j_{min} , j_{max} are the limits for parameter *j*. Can be any integer (positive, negative or zero).

The parameters required to define a grid are then $\mathbf{P}_0$, $\mathbf{V}_1$, $\mathbf{V}_2$ and the limits on *i* and *j* (i_{min} , i_{max} , j_{min} , j_{max}).

Be aware that marker grid primitives are subject to all transformations and can result in a large number of markers generated for a grid that is defined or transformed in three dimensions so it becomes parallel or near parallel to the direction of projection.

Line Grid Primitive

A line grid primitive is created using either the Line Grid 2 **(GPLG2)** or Line Grid 3 **(GPLG3)** subroutines and defines a planer grid of 2 groups of crossing line segments. The two-dimensional form of this primitive creates the grid in the XY plane (Z=0.0) of modeling coordinate space. The three-dimensional form specifies an arbitrary plane in three space in which the lines will be positioned.

A line segment is generated for each pair of endpoints (P_1 , P_2) for each value of *i* and *j* as shown in the following equations:

Line segments parallel to $\mathbf{V}_1$:

$$\mathbf{P}_1(j) = \mathbf{P}_0 + i_{min}\mathbf{V}_1 + j\mathbf{V}_2$$

$$\mathbf{P}_2(j) = \mathbf{P}_0 + i_{max}\mathbf{V}_1 + j\mathbf{V}_2$$

Line segments parallel to $\mathbf{V}_2$:

$$\mathbf{P}_1(i) = \mathbf{P}_0 + i\mathbf{V}_1 + j_{min}\mathbf{V}_2$$

$\mathbf{P}_2(i) = \mathbf{P}_0 + i\mathbf{V}_1 + j_{max}\mathbf{V}_2$

where:

$j_{min} <= j <= j_{max}$

$i_{min} <= i <= i_{max}$

$\mathbf{P}_1$, $\mathbf{P}_2$ = Endpoints of lines inthe grid.

$\mathbf{P}_0$ = a 2D or 3D modelling coordinate which defines the origin of the grid.

$\mathbf{V}_1$ = a 2D or 3D vector that defines the direction of the *i* grid lines and the relative spacing of the *j* grid lines.

$\mathbf{V}_2$ = a 2D or 3D vector that defines the direction of the *j* grid lines and the relative spacing of the *i* grid lines.

i_{min} , i_{max} = limits forparameter *i*. They can be any integer, positive, negative or zero.

j_{min} , j_{max} = limits forparameter *j*. They can be any integer, positive, negative or zero.

The parameters required to define a grid are then $\mathbf{P}_0$, $\mathbf{V}_1$, $\mathbf{V}_2$ and the limits on *i*and *j* (i(sub min), i(sub max), j(sub min), j(sub max)). This primitive is subject to all transformation and clipping and all polyline attributes are used in rendering it.

Be aware that a large number of lines may be generated for a grid that is defined or transformed in three space so that it becomes parallel or near parallel to the direction of projection.

Annotation Text Relative Primitive

The Annotation Text Relative primitive **(GPANR2** and **GPANR3)** define annotation text associated with a specified reference point. Using this primitive with the Set Annotation Style **(GPAS)** attribute, a line can be drawn from the reference point to the annotation text origin.

The transformation pipeline transforms the specified reference point to NPC coordinates. If the resulting point is clipped, then no annotation text is displayed. If the reference point is not clipped, then the annotation text is positioned relative to the reference point, as specified by the text vector. The rendering consists of the specified annotation text and an annotation style. Text is drawn using the offset specified by the text vector as the origin of the text coordinate system, and is drawn as normal annotation text using the current annotation traversal attribute values. Annotation style is controlled by the Set Annotation Style attribute. If the attribute is set to `LEAD_LINE`, a line is drawn from the reference point to the text origin, using the current polyline traversal attribute values. When drawn, the text string and the annotation style rendering are subject to the normal clipping rules (as long as the reference point is not clipped.)

Composite Fill Area Primitive

A composite fill area primitive is created by calling the Composite Fill Area 2 **(GPCFA2)** subroutine and is defined by the following line geometry forms:

- Line Segments
- Parametric Elliptical Arcs

  Note: Circular Arcs are a subset of these.
- Non-Uniform B-Spline Curves

The specified geometry segments define the contours of the fill area as shown in the following figure. The curves of each contour must be explicitly connected and each contour must be explicitly closed by connecting the last curve to the first. However, any gaps due to floating point inaccuracies will be closed implicitly.

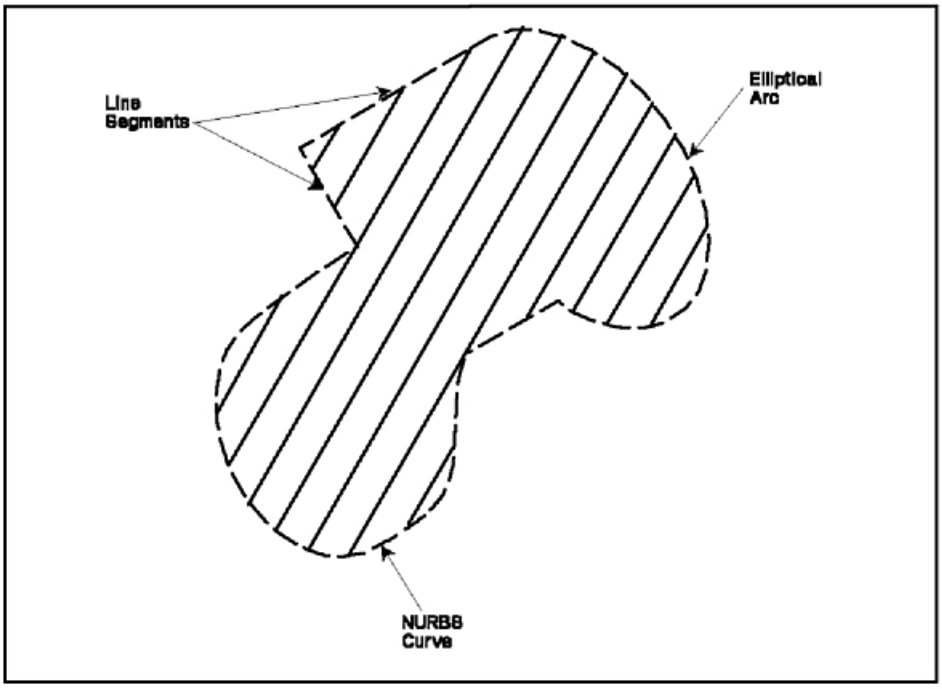


Figure 50. Composite fill area definition. This illustration shows a combination of various line geometry forms used to create a closed geometrical shape. The specific forms used in this illustration are line segments, elliptical arcs, and NURBS curves.

The polygon attributes including lighting and shading are used to render this primitive.

The composite fill area primitive may contain multiple loops in its boundary definition, similar to a polygon primitive.

The parameters your application must supply to the **GPCFA2** routine are:

- The number of contours
- A list of the number of curves in each contour
- An array containing the following information for each curve:
  - The type of curve (line, elliptical arc, NURBS curve, etc.)
  - A set of options for the curve. One option specifies whether the curve should be processed as an edge of the surface. Other options specify whether weights are specified with each control point and whether tessellation values are present.
  - The order of the basis function.
  - The number of points in the point list array to be used for this curve.
  - The parameter limits for the curve if the curve type requires them.
- A list of knots from which the knot vectors for non-uniform curves will be taken. The knot vectors must be placed in this list in the order that their curve definitions appeared in the previous array.
- The list of tessellation values for parametric curves whose options indicate they are present.
- The width of the point list array.
- The point list containing the coordinates and vectors for each of the curves.

The edge of a composite fill area consists of the set of boundary curves that define the region to be filled. Line styles applied to the edge are continuous over an entire contour.

The drawing direction of each segment of the primitive is important in order for the geometry segments to meet properly. For example, a NURBS curve is drawn from the minimum parameter value to the maximum parameter value. The geometry segments drawn before and after a NURBS curve must meet the curve at its start and end points respectively. In order to reverse the drawing direction of a NURBS curve your application must reverse the order of the control points and adjust the knot vector accordingly. To reverse the direction of an elliptical arc, see the definition of the elliptical arc primitive presented earlier in this chapter.

Polyhedron Edge Primitive

A polyhedron edge primitive is created by calling the Polyhedron Edge **(GPPHE)** subroutine and is used to represent the intersection of two planes or faces of a polyhedron type object. Multiple intersections may be specified within the same structure element. The definition of each edge segment requires the two end points of the edge and the two normals of the faces or planes whose intersection has created the edge.

The normals are used to control whether or not the line segment is actually rendered. When it is rendered, the polyline attributes are used.

A polyhedron primitive is used to model polyhedron objects in which some of the edges are considered sharp and others are only construction lines. This primitive can be used to produce various effects when modeling polyhedron objects as discussed in the Chapter 16. Rendering Pipeline.

Polysphere Primitive

A polysphere structure element is created using the Polysphere **(GPSPH)** subroutine. When drawn, a polysphere structure element will generate a set of spheres each defined by a center point and a radius. The polysphere primitive is an area defining primitive, therefore, polygon attributes and surface attributes apply to the interior of the polysphere. In addition, polyline attributes and surface attributes apply to the tesselation curves of the polysphere.

Because a sphere has no edges, polygon edge attributes do not apply to this primitive, and boundaries are not drawn. A sphere with Interior Style of `HOLLOW` does not have a direct effect on the display, except when HLHSR processing is active. During HLHSR processing, such a sphere is rendered into the Z-buffer, so a hollow sphere hides parts of other primitives that are farther away in their Z extent. Workstation-dependent techniques are used to perform highlighting color and pick echo for hollow and empty interior styles.

The polysphere primitive is ideal for such applications as molecular modeling and provide better performance than a NURBS surface shaped as a sphere.

Non-Uniform Relational B-Spline (NURBS) Primitive

The PHIGS PLUS standard and the associated graPHIGS API extensions allow curves and surfaces to be defined in terms of Non-Uniform Rational B-Splines (NURBS). With the appropriate choice of hardware architecture and software algorithms, rendering curved surfaces defined as NURBS is faster and more accurate than rendering the same surfaces defined by polygons.

There are a number of characteristics which must be understood in order to use NURBS curves and surfaces effectively. These include specification of the data representing curves and surfaces, as well as transformation, tessellation, and evaluation of NURBS curves and surfaces.

This section focuses on the characteristics and generation of NURBS entities from the graPHIGS API point of view. These entities consist of 2D parametric curves, 3D parametric curves, simple (untrimmed) parametric surfaces, and trimmed parametric surfaces. The following figure provides a simple example of a 2D NURBS curve:

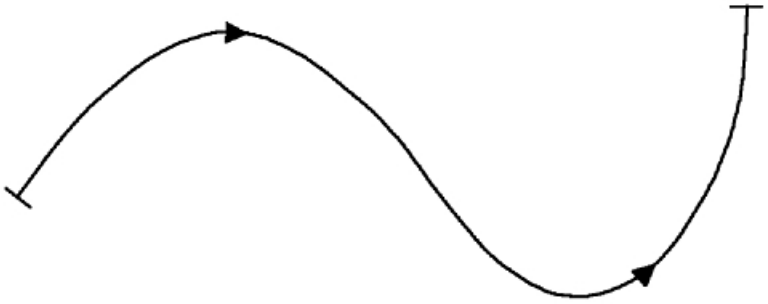


Figure 51. A 2D NURBS curve. This illustration depicts a sine curve as an example of a two-dimensional NURBS curve.

A NURBS surface consists of a 2D rectangle which may be bent, stretched, and twisted to form a 3-D shape. A simple or untrimmed NURBS surface, as shown in the following figure, consists of the entire rectangular surface which has been formed into a 3D shape.

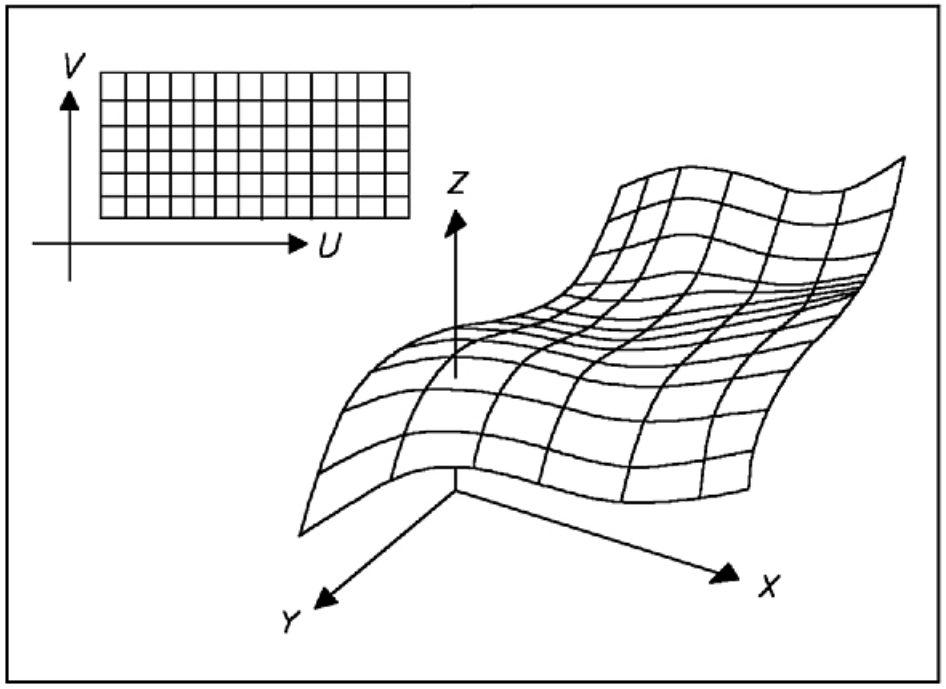


Figure 52. A NURBS Surface. This illustration shows two views of a rectangular grid. The first view shows a two-dimensional illustration of the grid. The second view shows the same grid bent, stretched, and twisted to form a three-dimensional shape.

A trimmed NURBS surface, as shown in the following figure, is similar to a dress pattern where the trimming curves define the pattern to be cut out of the surface.

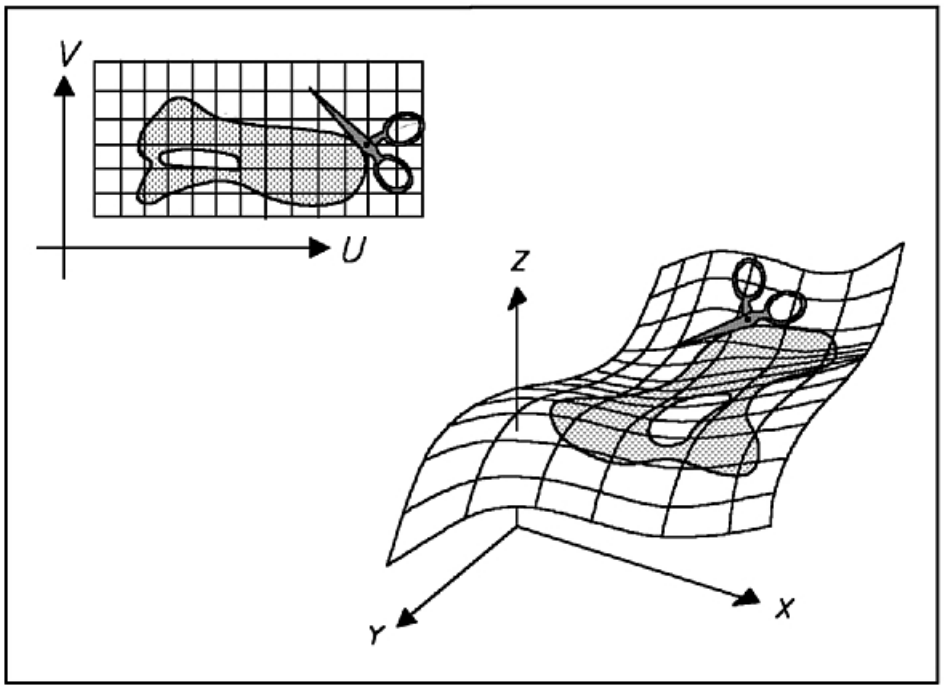


Figure 53. A Trimmed NURBS Surface. This illustration is similar to the one described in the previous figure. In this illustration, there is an enclosed, curved surface with an interior hole on the grid in both views. This surface is the trimmed NURBS surface.

One or more NURBS curves may be specified within the rectangular area which defines the surface. Any portions of this surface outside these curves are discarded. Anything bounded by an even number of curves, such as the interior of the hole shown in the figure, A NURBS Surface, is also discarded. The resulting 3D surface includes everything bounded by an odd number of curves.

Note: In the figure, ″A Trimmed NURBS Surface,″ the upper diagram represents trimming curves defined in the surface coordinates. The lower diagram represents the resulting 3D trimmed surface.

The following section reviews the essential characteristics of NURBS curves and surfaces. This will be followed by a discussion of tessellation, the process which controls the accuracy with which a curve or surface is rendered. Finally, we will examine how NURBS have been implemented as graPHIGS API primitives.

NURBS as Parametric Functions: In addition to the geometric coordinates (x,y,z), NURBS depend on one or two parametric coordinates (t, or u and v) which increase monotonically from the start of a curve to the end, or from one edge of a surface to the opposite edge. Thus, for 3D curves, we have:

$x = X(t)$

$y = Y(t)$ for $t_{min} <= t <= t_{max}$.

$z = Z(t)$

where: $X(t)$, $Y(t)$, and $Z(t)$ are functions of the parametric coordinate t. Likewise for surfaces, we have:

$x = X(u,v)$

$y = Y(u,v)$ for $u_{min} <= u <= u_{max}$,

$z = Z(u,v)$ and $v_{min} <= v <= v_{max}$.

In addition, for trimmed surfaces, the boundary curves are defined as:

$u = U(t)$

$v = V(t)$ for $t_{min} <= t <= t_{max}$.

NURBS are closely related to other types of parametric functions, such as Bezier functions, which are actually a special case of NURBS. Parametric curves and surfaces may also be defined in terms of simple polynomials, and any such parametric polynomial function may be represented exactly by a corresponding NURBS function.

NURBS as Piecewise Representation by Polynomials: NURBS curves and NURBS surfaces may be defined in either rational (hence the R in NURBS) or non-rational forms. In the simpler non-rational form, each component (x, y, or z) of a NURBS curve, for example, may be determined by evaluating a polynomial function of the parametric coordinate:

$$X(t) = \sum_{i=0}^{m} c_{xi}\ t^i$$

$$Y(t) = \sum_{i=0}^{m} c_{yi}\ t^i$$

$$Z(t) = \sum_{i=0}^{m} c_{zi}\ t^i$$

The rational case is similar, except that there is an additional function called the weight, which is defined as follows:

$$W(t) = \sum_{i=0}^{m} c_{wi}\ t^i \ .$$

In this case, the geometric coordinates are determined by ratios of polynomials:

X(t) = WX(t) / W(t),
Y(t) = WY(t) / W(t),
Z(t) = WZ(t) / W(t),

where: `WX(t)`, `WY(t)`, and `WZ(t)` are polynomials similar those which specify `X(t)`, `Y(t)`, and `Z(t)` for the non-rational case.

Each NURBS function, like any polynomial, may be characterized by the degree (*m*) corresponding to the highest power of the parametric coordinate. Equivalently, a NURBS function may be characterized by the order (*k=m+1*) corresponding to the number of linearly independent terms in a polynomial of degree *m* Thus a linear function has degree 1 and order 2, a quadratic function has degree 2 and order 3, and so forth.

Unlike a simple polynomial function, for which a single set of coefficients (c_i , $i = 0$ to m) characterizes the entire function, a NURBS function may be represented by multiple sets of coefficients, each of which is valid only for a limited range of the parametric coordinate. Thus, a curve may be divided into a sequence of spans, each represented by different polynomials. This is illustrated by the curve in the following figure which consists of three spans.

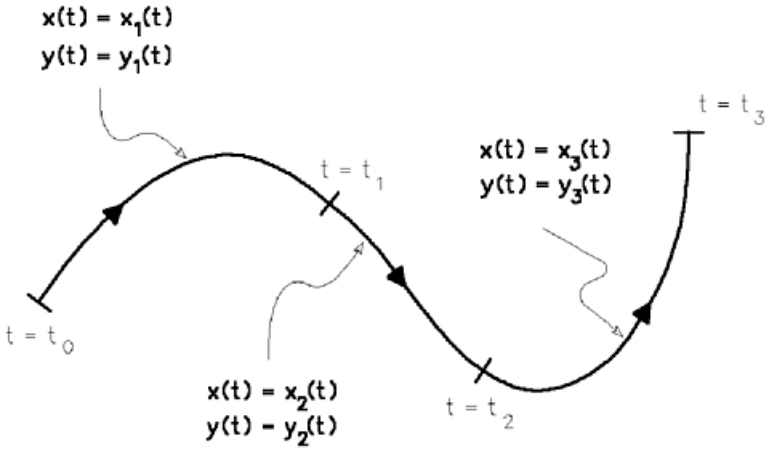


Figure 54. The piecewise polynomial representation of a curve. This illustration shows a curve made up of three spans. The first span of the curve uses $x(t)=x_1(t)$ and $y(t)=y_1(t)$ as its equations when t is between t_0 to t_1. The second span of the curve (between t_1 and t_2) uses $x(t)=x_2(t)$ and $y(t)=y_2(t)$. The third span (between t_2 and t_3) uses $x(t)=x_3$ and $y(t)=y_3$. The aforementioned equations are defined in the text that follows.

Note: This curve is determined by the six polynomials x_1, y_1, x_2, y_2, x_3, and y_3.

The first span is defined by parameter values t running from t_0 to t_1. The coordinates of this portion of the curve are determined by the following polynomials:

$$X_1(t) = \sum_{i=0}^{k} c_{x1i}\ t^i, \text{ and}$$

$$Y_1(t) = \sum_{i=0}^{k} c_{y1i} t^i.$$

The second span, defined by parameter values running from t_1 to t_2 , is determined by the following polynomials:

$$X_2(t) = \sum_{i=0}^{k} c_{x2i}\ t^i, \text{ and}$$

$$Y_2(t) = \sum_{i=0}^{k} c_{y2i}\ t^i.$$

Likewise, the third span, with parameter values running from t_2 to t_3 , is determined by the following polynomials:

$$X_3(t) = \sum_{i=0}^{k} c_{x3i}\ t^i, \text{ and}$$

$$Y_3(t) = \sum_{i=0}^{k} c_{y3i}\ t^i.$$

The parametric polynomials that describe adjacent spans of a NURBS curve are defined so as to provide a specific degree of continuity, where degree 0 means the values are continuous, degree 1 means the value and first derivative are continuous, and so forth. The maximum degree of continuity is *k-2* where *k* is the order of the functions. For example, a cubic curve (with degree *m=3* and order *k=4*) may have spans joining with continuity of degree 2 (continuous second derivatives). A lower degree of continuity may be obtained by constructing the knot vector as discussed in the next section.

NURBS Control Points, Weights, and Knot Vectors: NURBS are defined by control points, weights, and knot vectors. The control points provide the primary control over the geometry of a curve or surface. A curve may have *n* control points, where *n* must be greater than or equal to the order *k* of the curve. A surface has an n_u by n_v matrix of control points where $n_u >= k_u$ and $n_v >= k_v$, and k_u and k_v are the orders of the *u* and *v* parameters.

A NURBS curve or surface does not pass through (interpolate) the control points, except in certain special cases. Instead, the curve or surface merely passes near these points, as shown in the following figure in which the curve passes near the control points, but not through them. If needed, it is possible to construct a NURBS curve which interpolates a set of *n* points, and the resulting curve is determined by a set of *n* control points which is related to the *n* interpolated points by a simple *nxn* matrix.

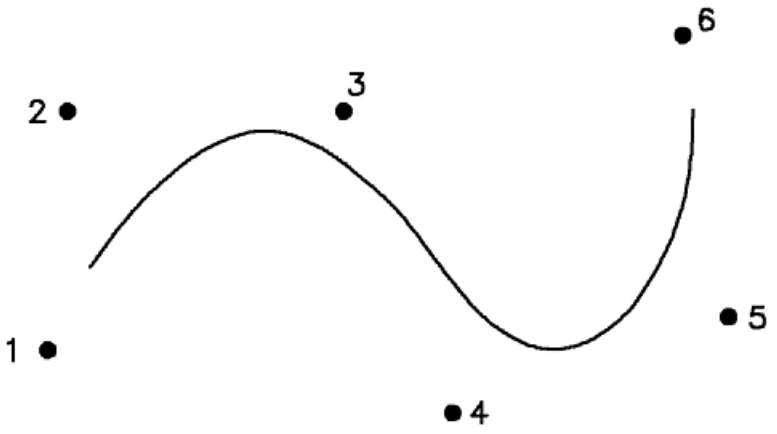


Figure 55. A NURBS curve and its control points. This illustration shows a curve that passes near six control points but does not go through them.

The weight values are optional and are used to define the rational form of the NURBS curves and surfaces. In this form, there is a weight value (*w*) associated with each of the control points (*x,y,z*). The

weights and coordinates are combined to form the homogeneous coordinates (WX, WY, WZ, W), which form the control points for a set of four parametric polynomial functions *WX*(*t*), *WY*(*t*), *WZ*(*t*), and *W*(*t*). The resulting geometric coordinates are determined by the following ratios:

$x(t) = WX(t) / W(t)$
$y(t) = WY(t) / W(t)$
$z(t) = WZ(t) / W(t)$

Similar expressions can be defined for surfaces. Consequently, each point is determined by a ratio of polynomials. In order to maintain a well behaved set of operations, the weights are required to be positive definite (must be greater than zero). The values of the weights are usually close to unity, and the rational form reduces to the non-rational form if the weights are all equal.

The knot vectors define a partitioning of the parameter space for the parametric coordinates t (for a curve), or *u* and *v* (for a surface). The knot vector for a curve of order *k* with *n* control points will have (*n+k*) components, of which the first and last are never used. Each span in the curve depends on *2 * m* successive knot values, where *m=k-1* and *k* is the order of the curve.

For a surface, there is a knot vector and a sequence of spans for each of two parametric coordinates. The portion of a surface determined by one span for each parameter is called a *patch*

If the values of the knots are spaced uniformly (for example, 0, 1, 2, 3, ...), then the result is called a uniform B-spline. For a non-uniform B-spline, the knot values may be separated by irregular intervals, and knot values may be repeated, as in the knot vector (0.0, 1.2, 1.5, 1.5, 2.7, 9.0).

As indicated previously, a curve of order *k* may be continuous up to degree *k-2*. A lower degree of continuity may be obtained by using repeated knot values in the knot vector. For example, a NURBS curve of order 4 (cubic) with no repeated knot values will be continuous first and second derivatives (degree 2 continuity). If a knot value is repeated so that the same value appears twice in a row in the knot vector, then the second derivatives will be discontinuous at the point where the parametric coordinate has the repeated value.

In this case, the function and first derivative would still be continuous. If a knot value is repeated twice (same knot value three times), then the first derivatives may be discontinuous, and the curve could have a sharp corner at this point. If a knot value occurs four times (or more), then the curve itself may be discontinuous.

The ability to match successive spans with a specified degree of continuity makes it possible to construct a complex curve passing through many points using low order NURBS. This property is extremely valuable because it makes it possible to avoid high order functions. High order functions, whether NURBS, simple polynomials, or any other type tend to be costly to evaluate and prone to numerical instabilities. High order polynomials often require double precision (64-bit) floating point operations for reliable evaluation, whereas low order NURBS may be evaluated with single precision (32-bit) arithmetic without loss of accuracy.

Characteristics of NURBS: NURBS curves and NURBS surfaces all share certain fundamental features. These features are responsible for the selection of NURBS as the means of representing curved lines and curved surfaces in the graPHIGS API As a tool for geometric modeling, NURBS provide the following:

- Local Shape Control
- Conic Sections and Quadric Surfaces
- Efficient Trivial Rejection Testing
- Efficient Evaluation Algorithms
- Invariance with Respect to Coordinate Transformations
- Efficient data compression

The following paragraphs provide a closer look at each of these characteristics. These paragraphs will use 2D curves to provide simple examples of each property. In most cases, the generalization to 3D curves and surfaces is very simple.

Local Shape Control: The local shape control means that each control point and each knot value influences or controls the shape of only a limited portion of a curve or surface. Each control point of a curve affects up to *k* spans, and each knot value affects up to *2 * m* spans, where *k* is the order of the curve and *m = k - 1* is the degree of the curve. Each control point for a surface with orders k_u and k_v affects a set of up to k_u x k_v patches.

For example, if the first control point for the curves is moved, then the first span ($t_2 < t < t_3$) will change, but the second span ($t_3 < t < t_4$) and the third span ($t_4 < t < t_5$) will not change.

The local shape control property may be compared to the global control found when fitting a single high order polynomial to a number of points. For example, consider a polynomial of order 12 that passes through a a set of 12 data points. If any point is altered, the entire curve changes. Changing the starting point will change the way the curve approaches the ending point. If a cubic NURBS curve is determined by a set of 12 control points, then the first control point will influence only a short section of the curve near the first point. The remainder of the curve does not depend on the location of the first point.

Conic Sections and Quadric Surfaces: NURBS can provide exact representations of the conic sections and quadric surfaces. The conic sections include circles and ellipses, as well as parabolas and hyperbolas. The quadric surfaces include the cylinder, cone, sphere, ellipsoids, hyperbolic paraboloid, and so forth. This class of curves and surfaces, especially the circle, cylinder, cone, and sphere, are essential for geometric modelling, especially in CAD/CAM applications. The exact representations of these shapes require the rational form of the NURBS with the associated weight values. This is the principal reason for generalizing parametric curves and surfaces to the rational form, and the principal reason for the support of these rational forms as part of the graPHIGS API.

The tables on Supported Primitives indicate how to specify the control points and weights for a circle. The geometry of these points is shown in the following figure. This is only one of an infinite number of ways to define a circle with NURBS. Similar sets of points and weights may be used to construct spheres, cylinders, and so forth.

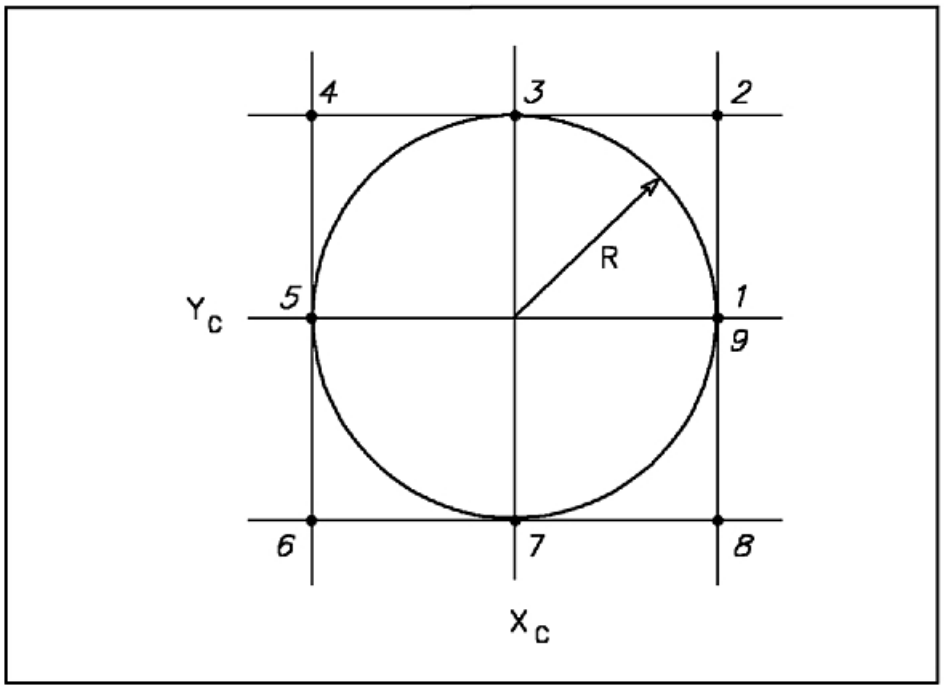


Figure 56. The control points for a circle. This illustration shows a circle drawn within a grid of four squares. Each intersection of the grid on the outer perimeter of the four squares is a control point. The points are given the weights 1 through 9 starting with the middle intersection on the far right side and traveling clockwise on the perimeter of the grid.

Note: In the figure, The control points for a circle, nine control points, along with the weight values specified in Table 1, are used to define a circle with radius R centered at (*Xc*, *Yc*). The knot vector for this curve is **(0,0,0,1,1,2,2,3,3,4,4,4).**

Table 5. The 2D circle as a NURBS curve. The following nine sets of control points and weights may be used to generate a circle with radius R centered at (Xc,Yc).

point	X-ctrl	Y-ctrl	weight
1	Xc + R	Yc	1
2	Xc + R	Yc + R	1/sqrt(2)
3	Xc	Yc + R	1
4	Xc - R	Yc + R	1/sqrt(2)
5	Xc - R	Yc	1
6	Xc - R	Yc - R	1/sqrt(2)
7	Xc	Yc - R	1
8	Xc + R	Yc - R	1/sqrt(2)
9	Xc + R	Yc	1

Efficient Trivial Rejection Testing: NURBS curves and NURBS surfaces are contained within the convex hull of their control points. The convex hull is the smallest convex polygon (2D case) or smallest convex polyhedron (3D case) which contains a given set of points.

This property applies to each span of a curve and each patch of a surface, as well as to a complete curve or surface. That is, each span of a curve must lie within the convex hull of the *k* control points which

define that span, where *k* is the order of the curve. Likewise, each patch of a surface is contained within the convex hull of a set of k_u by k_v points, where k_u is the order of the *u* parameter and k_v is the order of the *v* parameter.

This property makes it possible to make estimates of and place bounds on the size and location of a NURBS curve or surface without evaluating a single point. An important application of this property is found in trivial rejection testing. If the convex hull does not overlap with the current screen or window, then the surface is not evaluated. This has a large effect on the performance of a system when a user zooms in for a detailed view of a small portion of a complex scene or object. In this case, most of the surfaces can be rejected, allowing the display to be updated rapidly. Similar advantages are obtained when processing a NURBS curve or surface for pick input.

Efficient Evaluation Algorithms: There are several efficient algorithms for evaluating NURBS functions. In principle, the control points and knot values for a NURBS function may be used to determine the coefficients of the corresponding polynomials for each span. These may then be evaluated using Horner's Rule:

$$f(t) = \sum_{i=0}^{m} c_i t^i$$

$$= C_0 + t(C_1 + t(C_2 + \ldots))$$

This requires only *m* multiplications and *m* additions per point. Calculation of the coefficients, however, may cost more than evaluating a number of points. In addition, the coefficients in this sum frequently have large magnitudes with opposing signs. Consequently, the result may be very sensitive to round-off and truncation errors in the floating-point operations. In this case, the intrinsic numerical stability characteristic of NURBS has been destroyed by converting to the polynomial representation. Thus, even though NURBS may be formally equivalent to certain polynomials, it is usually best to avoid any explicit reference to these polynomials when processing a NURBS function.

An alternative to Horner's Rule is provided by the *forward difference method*. This is frequently used for Bezier curves which are a special case of NURBS. In this case, the control points are used to determine a set of forward difference components, d_i, which are related to successive derivatives of the curve. Each point is evaluated by adding each component to its lower order neighbor:

$d_i = d_i + d_{i+1}$, for $i = m-1, m-2, \ldots, 0$.

The resulting values of d_0 form successive values of the function. This requires only *m* additions, and no multiplications, making this a very efficient algorithm on a point-by-point basis. This efficiency is partly offset by the cost of computing the first set of forward difference components, d_i. In addition, each point is based on the value of the previous point, and is subject to the cumulative effects of the limited precision of the floating-point operations in all preceding points within a span. Although these errors will usually be small, they may lead to gaps between successive spans of a curve, or rips between patches of a surface.

A third algorithm, called the DeCasteljau or Cox-DeBoor algorithm, is based on recursively interpolating the control points, as shown in the following figure. Intermediate points A, B, and C are interpolated between control points 1, 2, 3, and 4 based on the parameter value and the knot vector. Intermediate points D and E are interpolated between points A, B, and C in a similar manner. Point F on the curve is interpolated between points D and E. Segment DE is tangent to the curve.

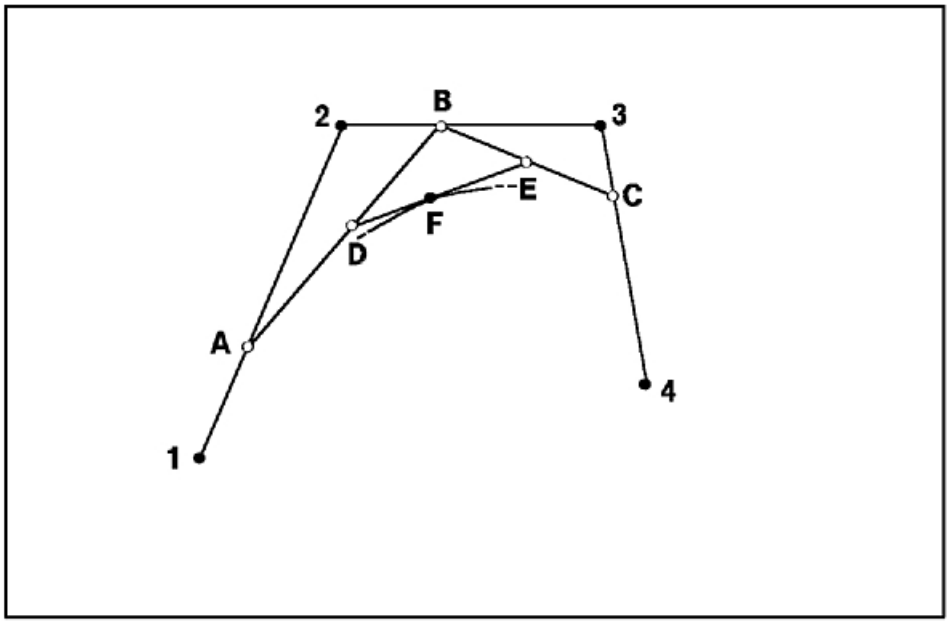


Figure 57. The Cox-DeBoor (DeCasteljau) algorithm for a cubic curve. This illustration shows four control points (1, 2, 3, and 4) and 6 intermediate points. The control points are displayed on a squared off curve created by three line segments. The first line segment is drawn from point 1 (lower left) to point 2 (upper left) to form a line that slants inward. The line segment from point 2 to point 3 forms a straight horizontal line. The last line segment is drawn from point 3 (upper right) to point 4 (lower right). This segment forms a lines that slants outward. Point A is on the line segment between points 1 and 2; B is between 3 and 3; and C is between 3 and 4. Two line segments connect points A, B, and C in the following way: one connects A and B, and the other connects B and C. Point D is on the line segment between A and B. Similarly, E is on the line segment between B and C. Another line segment connects points D and E. Point F is on this line segment and on part of the curve that line segment DE is tangent to.

One point on a cubic curve (order 4) would be given by the following operations:

$f_1 = ((a_1 - t)\ Q_1 + (t - b_3)\ Q_2) / (a_1 - b_3)$
$f_2 = ((a_2 - t)\ Q_2 + (t - b_2)\ Q_3) / (a_2 - b_2)$
$f_3 = ((a_3 - t)\ Q_3 + (t - b_1)\ Q_4) / (a_3 - b_1)$

$f_1 = ((a_1 - t)\ f_1 + (t - b_2)\ f_2) / (a_1 - b_2)$
$f_2 = ((a_2 - t)\ f_2 + (t - b_1)\ f_3) / (a_2 - b_1)$

$f(t) = ((a_1 - t)\ f_1 + (t - b_1)\ f_2) / (a_1 - b_1)$

where Q_1 ... Q_4 are four control point values, a_1, a_2, and a_3 are the three knots following the parameter t, b_1, b_2 , and b_3, are the three knots preceding the parameter t, and $f(t)$ is the result. The denominators are independent of t and the value of $1/(a_1 - b_1)$, for example, needs to be computed only once for each span.

This method requires on the order of m^2 floating-point operations, compared to order m operations for Horner's Rule and forward differences. This penalty is not severe for small m (e.g. $m = 3$ or cubic functions), and is largely offset by reduced set-up cost (the control points are used directly). The interpolation formulas preserve the numerical stability of the NURBS functions, and each point is computed independently, so there are no cumulative errors as the calculations progress across a span. Consequently the last point on a curve will be just as accurate as the first, and each span will match the next with maximum accuracy.

Invariance with Respect to Coordinate Transformations: A critical step in the processing of ordinary vectors and polygons is the transformation (rotation, translation, scaling, etc.) from Modelling Coordinates (MC) to screen coordinates or Device Coordinates (DC). Much attention has been devoted to developing specialized hardware for performing this process as efficiently as possible. With NURBS, such

transformations may be applied to either the points (vertices) generated by evaluation of the NURBS functions, or to the control points which define a NURBS curve or surface. In both cases the results are exactly the same. The number of control points, however, is usually much less than the number of vertices generated by the evaluation of the NURBS. Consequently, it is much more economical to transform the control points than to transform the resulting vertices.

Transforming the control points instead of the vertices has two additional advantages. First, the transformed control points may be used for trivial rejection based on the convex hull property. In addition, if the curve or surface is not rejected, then the number of vertices generated to represent a curve or surface will usually be much greater than the number of control points. Consequently, the processing of vertices, including evaluation of the NURBS, is more critical for performance than the transformation of the control points. In this case, the transformation of the control points may be pulled away from specialized transformation hardware and performed on a slower general purpose processor. The specialized processors normally used for transformations of vertices may then be used to accelerate the evaluation of the NURBS. This assumes that the specialized transformation hardware is not so highly specialized that it cannot be used for anything else.

Efficient data compression: NURBS curves and NURBS surfaces form very compact data structures for representing complex shapes. A typical NURBS surface may be determined by roughly 10 to 100 control points with 12 to 16 bytes of data for each control point depending on whether or not the rational form is being used. Thus, the total amount of data needed to define a NURBS surface may be in the range of 1k to 2k bytes.

If a graphics workstation (or display station) does not support NURBS, then the same surface may be approximated with a grid of polygons. The number of polygons required to represent a given surface depends on the accuracy or quality expected for the resulting images. This number must be sufficiently large to produce a satisfactory image at the largest scale at which the surface is to be displayed. These conditions may usually be satisfied with 100 to 1000 four-sided polygons. To support smooth shading, the coordinates (x, y, z) for each vertex must be accompanied by the corresponding surface normal (N_x, N_y, N_z), yielding a total of 24 bytes per vertex or roughly 100 bytes per polygon. As a result, it may require 10k to 100k bytes to approximate the same surface defined exactly by only 1k to 2k bytes of NURBS data.

As an example, a set of 17 NURBS surfaces can be used to closely approximate a jet aircraft. All together, these 17 surfaces require less than 18k bytes of data. When represented with polygons at a medium level of quality, the same surfaces require roughly 5000 4-sided polygons occupying over 500k bytes, or more than 25 times as much data as required by the NURBS surfaces.

The amount of data required to represent a surface as polygons may be reduced by a factor of from 2 to 4 by combining the polygons into more efficient data structures such as triangle strips or a quadrilateral mesh. On the other hand, the amount of data required for the polygons may also be increased by similar or larger factors by tightening the requirements for how well the polygons must approximate the true curved surfaces. Consequently, we see that the amount of data required to represent a object with NURBS is 10 to 100 times smaller than that required to represent the same object with polygons.

Because of the data compression inherent in NURBS, it is clear that NURBS are of great value as a data communications tool, as well as a graphics tool. Using NURBS instead of polygons is equivalent to increasing the host to workstation data transfer rate by a factor of 10 to 100. In addition, this same data compression factor means NURBS make much better use of system resources such as real memory and mass storage than the corresponding polygon data. This can translate into the ability to support more complex graphics, such as a comparison of ten different airplanes. In addition, this can influence performance by controlling whether a complex model resides mainly in cache, real memory, local mass storage, or even on a remote host.

Tessellation of Curves and Surfaces: Tessellation is the process by which a curved line is divided into segments that may be approximated with straight lines, or a curved surface is divided into areas that may be approximated with flat polygons. Ideally, tessellation would not be needed. A curved line, for example,

would be drawn as a sequence of points or pixels determined directly by the control points and/or other data defining the curved line. Lacking this capability, the workstation may divide a curved line into straight vectors (or anything else it can draw) in any manner as long as any resulting errors are negligible, such as less than one pixel.

Currently, however, there are several obstacles to this ideal. First, the judgement of what constitutes a negligible error is not uniquely defined. Secondly, even when a measure of error has been selected, as well as the size of this error which is considered to be negligible, the problem of determining how to satisfy these conditions can be more complex than the problem of evaluating the curve (or surface) at the selected points. Finally, the performance of current workstations tends to be strongly influenced by the size of the acceptable errors and the resulting number of points at which a curve or surface is evaluated.

The graPHIGS API permits the quality/performance trade-off to be controlled by allowing your application program to select how the error will be measured and the size of the acceptable error. In the case of NURBS curves, the measure of error is determined by the ″curve approximation criterion″, and the error tolerance is determined by a curve approximation control value. The recognized approximation criteria include:

- constant parametric division between knots
- metric (distance along a curve)
- chordal deviation

In the first case, the control value specifies the number of steps into which each span is to be divided. In the case of the metric criterion, the control value may indicate the maximum length of any segment in pixels. The chordal deviation is based on the distance between the true curve and the straight segments used to approximate the curve.

A similar set of criteria are defined for curved surfaces, except that there are two control values, one for each parametric coordinate. In addition, there is a set of corresponding criteria for the trimming curves for trimmed surfaces. These criteria are all defined as attributes, analogous to colors, line styles, polygon interior styles, and so forth.

The graPHIGS API supports the following three criteria for tessellation of curves and surfaces:

- `1=WORKSTATION_DEPENDENT`
- `3=CONSTANT_SUBDIVISION_BETWEEN_KNOTS`
- `8=VARIABLE_SUBDIVISION_BETWEEN_KNOTS`

The numbers (1, 3, and 8) were selected for consistency with PHIGS+.

Criterion 1 is based on the chordal deviation in Device Coordinates (DC). This method automatically adjusts the number of intervals in each span based on the curvature of the span and the scale of the figure. Criterion 1 is the criterion to use for all but a few unusual situations.

Criterion 3 specifies that each span (or each edge of each patch) is to be divided into a fixed number of intervals without regard to the curvature or scale of the figure. This criterion is provided for compatibility with PHIGS+.

Criterion 8 is an extension to the criteria defined in PHIGS+. This method permits an application to supply a tessellation vector which may represent any criterion an application may choose to define. Each component of the tessellation vector (specified as part of the data for a curve or surface) is proportional to the number of intervals for one span of a curve or one edge of a surface patch. This permits the tessellation to be custom-tailored by any application requiring exceptional treatment of curves or surfaces. This is the advanced option which would enable skilled programmers to substitute their own algorithm for the built-in algorithm used by the graPHIGS API for criterion 1.

In order to obtain the maximum performance, a tessellation algorithm will seek to evaluate a curve or surface at the minimum number of points required to satisfy the corresponding approximation criterion. In striving to meet this objective, however, a tessellation algorithm must also avoid the creation of ″artifacts″ such as pinholes (isolated missing pixels) or rips (sequences of missing pixels) in a surface. Such artifacts will occur, for example, if a surface is first divided into patches, and then each patch is tessellated independently. In this case, the vertices of the polygons which approximate the surface on one side of a patch boundary will not coincide with those on the other side. This creates small triangular holes along the boundaries between adjacent patches. The resulting artifacts can be very distracting when they permit a background of one color to show through holes in a surface of a sharply contrasting color. The API eliminates these artifacts by systematically tessellating each surface as a single entity.

Curve and Surface Primitives: The following paragraphs describe the support for NURBS curve and surface primitives within the graPHIGS API These structure elements are defined as Generalized Drawing Primitives (GDPs), and therefore may not be supported on all hardware platforms. To determine if the hardware your application is using supports NURBS curves and surfaces, use the Inquire List of Generalized Drawing Primitives **(GPQGD)** subroutine. The actual parameter limits of the subroutine calls discussed below should be inquired using the subroutines Inquire Curve Display Facilities **(GPQCDF)**, Inquire Surface Display Facilities **(GPQSDF)**, and Inquire Trimming Curve Display Facilities **(GPQTDF)**.

Non-Uniform Relational B-spline Curve 2/3: Two-dimensional and three-dimensional NURBS curve structure elements can be generated with the Non-uniform B-spline Relational Curve 2 **(GPNBC2)** and Non-uniform B-spline Curve 3 **(GPNBC3)** subroutines respectively. The parameters for both subroutines are discussed below:

order the order ($2 <= k <=$ max) of the B-spline basis function where 2 = linear, 3 = quadratic, 4 = cubic, etc. The maximum order (max) supported by a workstation can inquired using the Inquire Curve Display Facilities **(GPQCDF)** subroutine.

npoint number of control points (n) that must be greater than or equal to the order (k) of the curve. The maximum number of control points is limited by the maximum structure element size.

knot knot vector with $n + k$ values = (t_1, t_2, ..., $t_{(n + k)}$) The knot vector is subject to the following restrictions:

- the values must form a non-decreasing sequence:

  $t_i >= t_{(i - 1)}$
- the first and last values are not used but must still be non-decreasing.

The knot vector has the following properties:

- results unaffected by shifting: $t_i > t_i + x$ for any x
- results unaffected by scaling: $t_i > t_i$ x s for any s

Typically, you should choose the following values for the first two and last two knots:

- $t_1 = t_2 = 0.0$
- $t_{(n + k)} = t_{(n + k - 1)} = 1.0$ or the number of non-degenerate spans.

tflag tessellation data flag where:

0 indicates that no tessellation data is supplied

1 indicates that tessellation data is supplied for curve approximation criterion 8

tdata tessellation data vector of $n - k + 1$ values equal to (f_1, f_2, ...) If the curve approximation criteria is 8 then the i_{th} span will be divided in to Q x f_i intervals where Q is the control value supplied with the curve approximation criterion.

Note: If the curve approximation criteria is 1 and tessellation data is supplied, performance will be hindered due to the storage overhead of the tessellation data.

cflag control point flag which indicates whether or not the NURBS curve is non-rational (*cflag* = 0) or rational (*cflag* = 1). For the non-rational case, no weight information will be supplied in the *cltpts* array.

cwidth the number of values between subsequent *x* values in the *ctlpts* array. For **GPNBC2** *cwidth* must be greater than or equal to *cflag* + 2. For **GPNBC3** *cwidth* must be greater than or equal to *cflag* + 3.

ctlpts a *npoint* by *cwidth* control point array which contains the principal geometric data for the curve. For the rational form of the curve (*cflag* = 1) each control point is defined as (x_i, y_i, w_i) for **GPNBC2** and (x_i, y_i, z_i, w_i) for **GPNBC3**

tmin minimum parameter value

tmax maximum parameter value

The parameters *tmin* and *tmax* are subject to the following restrictions:

tmin >= t_k and *tmin* <*tmax*

tmax <= $t_{(n + 1)}$ and *tmax* >*tmin*

To obtain a complete curve set *tmin* = t_k and *tmax* = $t_{(n + 1)}$.

The number of spans generated by **GPNBC2** and **GPNBC3** will be *n* - *k*. If fewer than *k* + 1 control points are specified, no curve will be generated.

Non-Uniform Relational B-Spline Surface: A NURBS surface can be defined using the Non-Uniform B-Spline Surface **(GPNBS)** subroutine. The parameters for **GPNBS** are defined as follows:

uorder the order (2 <= k_u <= max) of the B-spline basis function for the surface in the *u* direction where 2 = linear, 3 = quadratic, 4 = cubic.

vorder the order (2 <= k_v <= max) of the B-spline basis function for the surface in the *v* direction where 2 = linear, 3 = quadratic, 4 = cubic.

The maximum surface order (*max*) supported can be inquired using the Inquire Surface Display Facilities **(GPQSDF)** subroutine.

unum number of control points in the *u* direction (n_u)

vnum number of control points in the *v* direction (n_v)

uknots vector of $n_u + k_u$ knot values = (u_1, u_2, ..., $u_{(nu + ku)}$)

vknots vector of $n_v + k_v$ knot values = (v_1, v_2, ..., $v_{(nv + kv)}$)

A knot vector is subject to the following restrictions:

- the values must form a non-decreasing sequence: $u_i >= u_{(i - 1)}$ and $v_i >= v_{(i - 1)}$
- the first and last values are not used but must still be non-decreasing.

The knot vector has the following properties:

- results unaffected by shifting: u_i —> $u_i + x$ and v_i —> $v_i + x$ for any x
- results unaffected by scaling: u_i —> u_i x s and v_i —> v_i x s for any s

Typically, you should choose the following values for the first two and last two knots:

- $u_1 = u_2 = v_1 = v_2 = 0.0$
- $u_{(nu + ku)} = u_{(nu + k_{u - 1})} = v_{(nv + kv)} = v_{(nv + k_{v - 1})} = 1.0$

tflag tessellation data flag where:

0 indicates that no tessellation data is supplied

1 indicates that tessellation data is supplied for surface approximation criterion 8.

utdata tessellation data vector in the *u* direction containing $n_u - k_u + 1$ values equal to $f_{u1}, f_{u2}, ..., f_{u(nu - k_u + 1)}$.

vtdata tessellation data vector in the *u* direction containing $n_v - k_v + 1$ values equal to $f_{v1}, f_{v2}, ..., f_{v(nv - k_v + 1)}$.

If the surface approximation criteria is 8 then the i^{th} *u* span will be divided into Q_u x f_{ui} intervals and the *v* span will be divided into Q_v x f_{vi} intervals where Q_u and Q_v are the control values supplied with the surface approximation criterion.

Note: If the surface approximation criteria is 1 and tessellation data is supplied, the performance will be hindered due to the storage overhead of the tessellation data.

cflag control point flag that indicates whether or not the NURBS surface is non-rational (*cflag* = 0) or rational (*cflag* = 1). For the non-rational case, no weight information will be supplied in the *cltpts* array.

cwidth the number of values between subsequent *x* values in the *ctlpts* array. Note that *cwidth* must be greater than or equal to *cflag* + 3.

ctlpts a *unum* by *vnum* by *cwidth* control point array which contains the principal geometric data for the surface. For the rational form of the surface (*cflag* = 1) each control point is defined as (x_i, y_i, z_i, w_i)

umin minimum *u* parameter value

umax maximum *u* parameter value

vmin minimum *v* parameter value

vmax maximum *v* parameter value.

The parameters *umin*, *umax*, *vmin*, and *vmax* are subject to the following restrictions:

- *umin*>= u_{ku} and *umin* < *umax*
- *umax*<= $u_{(n_u + 1)}$ and *umax* > *umin*
- *vmin*>= v_{kv} and *vmin* < *vmax*
- *vmax*<= $v_{(n_v + 1)}$ and *vmax* > *vmin*

To obtain the complete surface, set *umin* = u_{k_u}, *umax* = $u_{(n_u + 1)}$, *vmin* = v_{kv}, and *vmax* = $v_{(n_v + 1)}$. The number of surface patches generated by **GPNBS** will be *unum* - *uorder* by *vnum* - *vorder*. If *unum* or *vnum* is less than or equal to the order for the corresponding direction, no patches will be generated.

The edge for this primitive consists of the lines ofconstant parameter at the boundary of the surface.

Trimmed Non-Uniform Relational B-Spline Surface: Trimmed Non-Uniform Relational B-Spline surface primitives are generated using the Trimmed Non-Uniform Relational B-Spline Surface **(GPTNBS)** subroutine. The parameters for **GPTNBS** includes all the parameters of the Non-Uniform Relational B-Spline Surface primitive except the *u* and *v* parameter limits are specified as a set of curves comprising multiple contours called **trimming curves** The contours can be disjoint and can be contained inside one another to produce surfaces with holes. Trimming curves are similar to the edges of a polygon except that polygon edges are defined in three-dimensional modeling space while trimming curves are defined in the two-dimensional parameter space (*u*, *v* coordinates) of the surface.

Trimming curves may be any two dimensional non-uniform realational B-spline curve which adheres to the following guidelines:

- a mix of rational and non-rational curves may be used on the same surface

- the trimming curves may form multiple closed loops
- each loop must be explicitly closed
- each curve is parameterized independently
- trimming curves must not go outside the parameter limits of the surface
- the trimming curves within a loop must be connected in a head to tail fashion and they may not be randomly specified
- the trimming curves may not cross other trimming curves in the same or different loop

If these guidelines are violated, the resulting display will be indeterminate.

The interior or ″rendered″ part of the surface is defined by the applying the *odd winding rule* within the its parameter space. This is the same rule used by the graPHIGS API to determine the interior of polygons.

In addition to the parameters specified for the **GPNBS** subroutine, the following information is required to specify trimming curves:

ncontour
: number of trimming curves (0)

ncurve array of *ncontour* entries specifying the number of curves in each contour (nc_1, nc_2, ..., $_{ncontour}$) where each nc_i <0

curve-data
: array of curve data defined as follows:
  - one data record for each curve
  - number of data records = sum of *ncurve* values, nc_i
  - each record has 6 values:
    1. type of curve (always 3, NURBS curve)
    2. options (flags for tessellation, rational, and boundary conditions)
    3. curve order (integer, k_t = 2...*max*)
    4. number of control points (*npoint*)
    5. starting parameter value (*tmin*)
    6. ending parameter value (*tmax*)

tknot knot vectors of all trimming curves concatenated into one vector with *npoint* + k_t values per curve

ttess tessellation data for of all trimming curves with the optional tessellation flag turned ON. The data is concatenated into one vector with *npoint* - k_t - 1 values per curve. If the optional tessellation flag is `OFF` for all curves then this parameter is treated as a place holder in the **GPTNBS** subroutine call. Tessellation data should *not* be specified if the trimming curve approximation criteria is set to 1.

cdwidth
: number of words of data supplied with each control point

cddata array of control point data records for each trimming curve. The number of records equals the sum of all *npoint* values and each record has *cdwidth* data values (u_i, v_i, w_i, ...) where the w_i terms are present only for rational trimming curves. All trimming curves which are specified to be part of the edge will be rendered using edge attributes after the interior has been processed.

Primitive Attribute Elements

This section contains a discussion of the attributes applied to various primitives. The attributes are discussed according to the primitive group to which they apply. These groups are:

- Attributes Applied to All Primitives
- Polyline Attributes

- Polymarker Attributes
- Geometric and Annotation Text Attributes
- Polygon Attributes

Each group is further divided into *basic* and *advanced* attributes. The basic attributes will only be described briefly as they are fully explained in Chapter 4. Structure Elements.

Many of the attributes affect the color of primitives. There are two ways to define attribute colors; by an index into a color table or directly by specifying a color vector. A color vector consists of three color values that are interpreted based on the current direct color model. Your application can set the direct color model using the Set Direct Color Model **(GPDCM)** subroutine. For a complete discussion of the color pipeline, see Chapter 17. Manipulating Color and Frame Buffers.

Attributes Applied to All Primitives

Basic Primitive Attributes

The following attributes are discussed in detail in Chapter 4. Structure Elements.

Highlighting Color Index: The Set Highlighting Color Index **(GPHCI)** subroutine creates a structure element which sets the highlighting color for all subsequent primitives by specifying an index into the workstation's rendering color table.

Add Class Name to Set: The Add Class Name to Set **(GPADCN)** subroutine creates a structure element which adds the specified class names to the current class name set. All primitives in the structure created after a call to **GPADCN** will have the class name set as an attribute. Class names are integer identifiers used to make primitives invisible, highlighted, and pickable.

Remove Class Name from Set: The Remove Class Name from Set **(GPRCN)** subroutine creates a structure element which removes the specified class names from the current class name set.

Pick Identifier: The Set Pick Identifier **(GPPKID)** subroutine creates a structure element which sets the current pick identifier for all subsequent primitives. Pick identifiers are part of the pick path information returned by a pick input device. For more information on pick devices and pick paths, see Chapter 7. Input Devices and Chapter 18. Advanced Input and Event Handling.

Advanced Primitive Attributes

Highlighting Color Direct: The Set Highlighting Color Direct **(GPHLCD)** subroutine creates a structure element which sets the highlighting color for all primitives. (See Frame Buffer Comparison Options for a discussion of line-on-line highlighting used as a special rendering effect.) This attribute element is similar to the Highlighting Color Index element except that the color is specified with a color vector instead of an index into the rendering color table.

Color Processing Index: The Set Color Processing Index **(GPCPI)** subroutine creates a structure element which sets the current color processing method by specifying an index in the workstation's color processing table. (See Frame Buffer Comparison Options for a discussion of line-on-line highlighting used as a special rendering effect.) An application can use the Inquire Color Processing Facilities **(GPQCPF)** subroutine to determine the number of color processing method table entries available. For a complete description of color processing, refer to Chapter 17. Manipulating Color and Frame Buffers and Chapter 16. Rendering Pipeline.

Depth Cue Index: The Set Depth Cue Index **(GPDCI)** subroutine creates a structure element which sets the current depth cue information by specifying an index into the workstation's depth cue table. An application can use the Inquire Depth Cue Facilities **(GPQDCF)** subroutine to determine the number of entries in the depth cue table. For a complete description of depth cueing, refer to Chapter 16. Rendering Pipeline.

HLHSR Identifier: The Set HLHSR Identifier **(GPHID)** subroutine creates a structure element which sets the current Hidden Line / Hidden Surface Removal (HLHSR) mode. The valid HLHSR identifiers are:

```
1=VISUALIZE_IF_NOT_HIDDEN
2=VISUALIZE_IF_HIDDEN
3=VISUALIZE_ALWAYS
4=NOT_VISUALIZE
5=FACE_DEPENDENT_VISUALIZATION
6=NO_UPDATE
7=GREATER_THAN
8=EQUAL_TO
9=LESS_THAN
10=NOT_EQUAL
11=LESS_THAN_OR_EQUAL_TO
```

For a complete description of Hidden Line/Hidden Surface Removal, refer to Chapter 16. Rendering Pipeline.

Antialiasing Identifier: The Set Antialiasing Identifier **(GPAID)** subroutine creates a structure element that indicates whether antialiasing is to be performed by the workstation. Antialiasing reduces the jagged appearance of objects drawn on raster displays by using techniques that ″blur″ the pixels to a more uniform appearance. Antialiasing techniques are workstation dependent, and not all workstations provide antialiasing capability. Refer to the workstation description information in *The graPHIGS Programming Interface: Technical Reference* for a description of available antialiasing methods and restrictions. Use the Inquire Available Antialiasing Mode **(GPQAMO)** to determine the antialiasing capabilities of your workstation. The application may control the antialiasing identifier attribute using the Set Extended View Representation **(GPXVR)** subroutine. This subroutine lets you specify that the antialiasing identifier be ignored (`1=OFF`) or applied (`2=SUBPIXEL_ON_THE_FLY`) or (`3= NON_SUBPIXEL_ON_THE_FLY`) in the specified view.

Set View Index: The Set View Index **(GPVWI)** subroutine creates a structure element which changes certain current traversal values to those specified in the view table entry. This function provides compatibility with the ISO PHIGS standard. For best results, use this subroutine only if your application is already written to the ISO PHIGS standard, and avoid it if your application uses the graPHIGS subroutines for viewing (e.g. Associate Root To View **(GPARV)** and Set View Priority **(GPVP)**.

Polyline Attributes

The polyline attributes apply to the following primitives:

- Polyline 2/3
- Disjoint Polyline 2/3
- Polyhedron Edge
- Line Grid 2/3
- Circle 2
- Circular Arc 2
- Ellipse 2/3
- Elliptical Arc 2/3
- Non-Uniform B-Spline Curve 2/3

Basic Polyline Attributes

The basic polyline attributes, defined in detail in the first part of this guide, are reviewed in the following paragraphs.

Polyline End Type: The Polyline End Type **(GPPLET)** subroutine creates a structure element which sets the polyline end type to either flat, round, or square.

Polyline Index: The Set Polyline Index **(GPPLI)** subroutine creates a structure element which is an index into a workstation polyline bundle table.

Polyline Linetype: The Set Linetype **(GPLT)** subroutine creates a structure element which sets the polyline line type by specifying an index in the workstation's line type table.

Linewidth Scale Factor: The Set Linewidth Scale Factor **(GPLWSC)** subroutine creates a structure element which sets the polyline line width scale factor. The line width scale factor attribute is used to define the width of polyline primitives as a fraction of the workstation's nominal line width.

Polyline Color Index: The Set Polyline Color Index **(GPPLCI)** subroutine creates a structure element which sets the polyline color by specifying an index in the workstation's rendering color table. For more information on the processing of colors, see Chapter 17. Manipulating Color and Frame Buffers.

Attribute Source Flags: The Attribute Source Flag Setting **(GPASF)** subroutine creates a structure element which sets the attribute source flags for polyline line type, line width scale factor, and color attributes. Each attribute source flag controls whether a specified polyline attribute will be selected from individual attribute elements or through the workstation's polyline bundle table.

Advanced Polyline Attributes

The advanced attributes applied to polyline primitives include the following:

Polyline Color Direct: Set Polyline Color Direct **(GPPLCD)** creates a structure element which sets the line color for all polyline primitives. This attribute element is similar to Set Polyline Color Index **(GPPLCI)** except that the color is specified with a color vector instead of an index into the rendering color table.

Polyhedron Edge Culling Mode: The Set Polyhedron Edge Culling Mode **(GPPHEC)** subroutine creates a structure element which controls the display of polyhedron edges. (This attribute applies only to the Polyhedron Edge **(GPPHE)** primitive.) Polyhedron edges are either front-facing or back-facing, depending on the transformed normals specified in the Polyhedron Edge primitive. You can control culled (not displayed) edges, using different settings of the culling mode attribute. You can specify culled edges if both normals are back-facing (`2=BOTH_BACK`), front-facing (`3=BOTH_FRONT`), facing the same direction (`4=BOTH_BACK_OR_FRONT`), face different directions (`5=BACK_AND_FRONT`), or if one normal is back-facing (`6=LEAST_ONE_BACK`) or front-facing (`7=LEAST_ONE_FRONT`). The Inquire Advanced Attributes Facilities **(GPQAAF)** subroutine returns which polyhedron edge culling modes are supported on a specified workstation.

Vertex Morphing: Morphing modifies the geometry and/or data (*texture*) mapping data of a primitive. There are two types of morphing, one using vertex coordinates and the other using data mapping data values. These functions are called *vertex morphing* and *data morphing*, respectively. Because data morphing applies to area primitives only, data morphing cannot be used for polylines. Vertex morphing takes place in modeling coordinates. You define the changes using morphing values at each vertex of the affected primitives along with scale factors that define how the vertex morphing values affect the primitive. Use the Set Vertex Morphing Factors **(GPVMF)** subroutine to specify the scale factors. For more detailed information on vertex morphing, see Morphing. Use the Inquire Workstation Description **(GPQWDT)** call to determine whether your workstation supports the graPHIGS API morphing facilities.

Modeling Clipping: The modeling clipping function provides a way to clip geometric entities in world coordinates. The clipping volume in world coordinates is set by the Set Modeling Clipping Volume 2/3 **(GPMCV2)** and **(GPMCV3)** elements. Modeling clipping is activated or deactivated by the Set Modeling Clipping Indicator **(GPMCI)** element. Use the Inquire Workstation Description **(GPQWDT)** subroutine to determine whether modeling clipping is supported on your workstation. See Modeling Clipping for more information.

Curve Approximation Criteria: The Set Curve Approximation Criteria **(GPCAC)** subroutine creates a structure element which sets the curve approximation criteria for subsequent NURBS curves. **GPCAC** controls the tessellation of curves and applies only to the Non-Uniform B-Spline Curve 2/3 primitives.

Tessellation is the division of each span into intervals which can be drawn as straight lines. The number of intervals used to represent each span is determined by the criterion and control value as summarized below:

(Criterion 1) `WORKSTATION_DEPENDENT`: The workstation-dependent curve approximation criterion relies on a control value *Q* which represents the quality of the curve approximation where 1.0 is nominal quality. A value of *Q* greater than or less than 1.0 produces a higher or lower quality curve respectively. *Q* is used to determine the number of tessellation intervals (*n*). The 6090 workstation uses a technique called *chordal deviation* to determine how close to approximate the curve. Chordal deviation is the perpendicular distance between the actual curve and the approximating line segment as shown in the following figure. Chordal deviation is independent of modeling transformation.

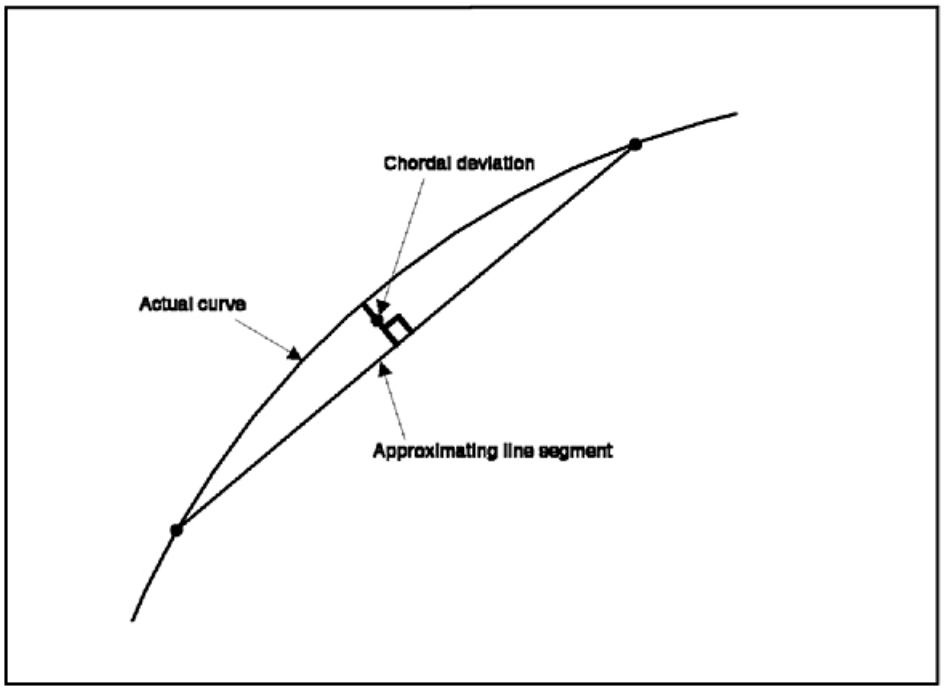


Figure 58. Definition of chordal deviation. This illustration depicts a curve and a chord; that is, a line segment that connects two points on the curve. The chord is referred to as the approximating line segment in this illustration and the curve is called the actual curve. The chordal deviation is the perpendicular distance between the curve (approximating line segment) and the actual curve.

(Criterion 3) `CONSTANT_SUBDIVISION_BETWEEN_KNOTS`: The simplest curve approximation criterion involves specifying a the number of subdivisions between knots as a constant *N*

(Criterion 8) `VARIABLE_SUBDIVSION_BETWEEN_KNOTS`: In the case of criterion 8, the number of tessellation intervals (*n*) is equal to the following: $n = Q \times f_i$

where:

Q is the tessellation control value

f_i is the i^{th} tessellation data value supplied with the definition of the NURBS curve.

Bundle Tables

In addition to individual attributes structure elements, polyline attributes may also be controlled by bundle tables. The Set Extended Polyline Representation **(GPXPLR)** subroutine is an advanced version of the Set Polyline Representation **(GPPLR)** subroutine. Both subroutines allow your application to define bundle

table entries for polyline attributes. Specifically, with **GPXPLR** your application can define polyline bundle table colors directly as color vectors as well as with a color index. To determine the number of entries in the bundle table, your application may use the Inquire Actual Length of Workstation State Tables **(GPQALW)** subroutine.

Application Defined Line Types

The Set Line Type Representation **(GPLTR)** subroutine allows your application to define line type entries in the workstation's line type table. To determine the number of entries in the line type table, your application can use the Inquire Polyline Facilities **(GPQPLF)** subroutine.

Note: Entry number one, `SOLID_LINE`, cannot be changed.

Your application controls the way a line representation renders a line by using the Set Linetype Rendering **(GPLNR)** subroutine. Currently, the graPHIGS API supports two rendering styles:

1=WORKSTATION_DEPENDENT_RENDERING

Typically, this line pattern is used to render the line, regardless of whether the end of the pattern falls on the endpoint of the line.

2=SCALED_TO_FIT_RENDERING

The `SCALED_TO_FIT_RENDERING` line rendering style renders a line using an integral number of repetitions of the line pattern. This is achieved by slightly scaling the pattern with each repetition. This scaling occurs in Device Coordinates, therefore, the line pattern is applied after transformation and clipping. In addition, you can specify a minimum line size for display. For example, if the resulting line is less than the minimum size, then the line is rendered as a solid line. The line pattern is applied from point to point in the line, and for disjoint polylines the pattern applies to each draw section of the disjoint polyline.

There are some restrictions for applying patterns using the `SCALED_TO_FIT_RENDERING` rendering style. First, it applies only to line primitives using Polyline attributes (except for the `ISOPARAMETRIC_LINES` of surfaces), and does not apply to polygon edges. Also, the `SCALED_TO_FIT_RENDERING` style does not apply to shaded lines, or take HLHSR into account. In addition, note that linewidth scale factor is ignored (only the nominal linewidth is used).

Line rendering styles are workstation dependent. To determine your workstion's supported styles, use the Inquire List of Line Rendering Styles **(GPQLNR)** subroutine to determine the supported rendering styles.

Polymarker Attributes

The polymarker attributes apply to the following primitives:

Polymarker 2/3
Marker Grid 2/3

Basic Polymarker Attributes

The basic polymarker attributes defined in the first part of this guide include the following:

Polymarker Index: The Set Polymarker Index **(GPPMI)** subroutine creates a structure element which represents and index into a workstation's polymarker bundle table.

Marker Type: The Set Marker Type **(GPMT)** subroutine creates a structure element which sets the polymarker marker type by specifying an index in the workstation's marker type table.

Marker Size Scale Factor: The Set Marker Size Scale Factor **(GPMSSC)** subroutine creates a structure element which sets the polymarker marker size scale factor during structure traversal. The actual size of a polymarker is the product of the nominal marker size and the marker size scale factor.

Polymarker Color Index: The Set Polymarker Color Index **(GPPMCI)** subroutine creates a structure element which sets the polymarker color by specifying an index in the workstation's rendering color table. For more information on color tables and the processing of color indexes, see Chapter 17. Manipulating Color and Frame Buffers.

Attribute Source Flags: The Attribute Source Flag Setting **(GPASF)** subroutine creates a structure element which sets the attribute source flags for polymarker marker type, marker size scale factor, and color attributes. Each attribute source flag indicates whether the attribute will be control by individual attribute structure elements or bundle table entries.

Advanced Polymarker Attributes
The following advanced attribute is defined for polymarkers:

Polymarker Color Direct: The Set Polymarker Color Direct **(GPPMCD)** creates a structure element which sets the color for all polymarker primitives. This attribute element is similar to Set Polymarker Color Index **(GPPMCI)** subroutine except that the color is specified with a color vector instead of an index into the rendering color table. For more information on direct colors, see Chapter 17. Manipulating Color and Frame Buffers.

Bundle Tables
In addition to individual attributes structure elements, polymarker attributes may also be controlled by bundle tables. The Set Extended Polymarker Representation **(GPXPMR)** is an advanced version of the Set Polymarker Representation **(GPPMR)** subroutine. Both subroutines allow your application to define bundle table entries for polymarker attributes. **GPXPMR** allows your application to define polymarker colors directly as well as with a color index. To determine the number of entries in the bundle table, your application may use the Inquire Actual Length of Workstation State Tables **(GPQALW)** subroutine.

Application Defined Marker Types
The Set Marker Type Representation **(GPMTR)** subroutine allows your application to define marker type entries in the workstations marker type table. The marker type representation is specified as a set of move and draw commands defined in the marker's coordinate system. To determine the number of entries in the marker type table, your application can use the Inquire Polymarker Facilities **(GPQPMF)** subroutine.

Note: Entry number three, ASTERISK, cannot be changed.

Geometric and Annotation Text Attributes
The text attributes discussed in this section apply to the following primitives:

Geometric Text 2/3
Annotation Text 2/3

Basic Geometric and Annotation Text Attributes
The following basic attributes are discussed in detail in Chapter 4. Structure Elements.

Text Alignment: The Set Text Alignment **(GPTXAL)** subroutine creates a structure element which defines the alignment point for all subsequent geometric text primitives relatives to the primitive's extent rectangle.

Text Precision: The Set Text Precision **(GPTXPR)** subroutine creates a structure element which controls the rendering detail of geometric and annotation text primitives. The maximum text precision supported on a workstation can be obtained using the Inquire Geometric Font Characteristics **(GPQGFC)** and Inquire Extended Annotation Font Characteristics **(GPQXAF)** subroutines. The support of specific text attributes based on text precision is documented in the *The graPHIGS Programming Interface: Subroutine Reference*, under the **GPTXPR** subroutine.

Character Height: The Set Character Height **(GPCHH)** subroutine creates a structure element that sets the height of the nominal character box (h_n) for geometric text primitives. Character box height is measured from the baseline to the capline along the character up vector as shown in the following figure:

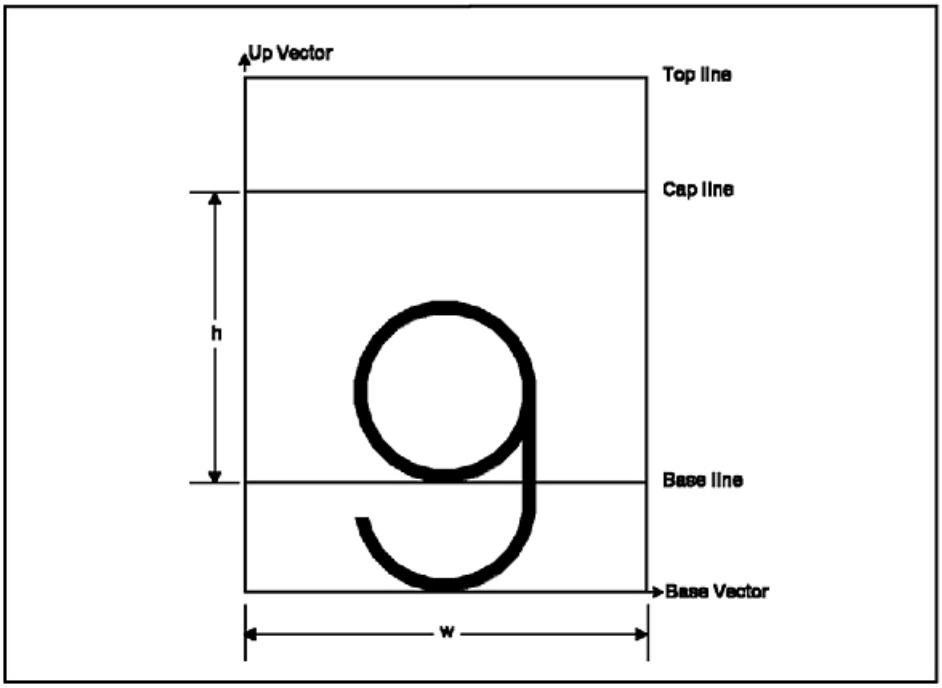


Figure 59. Example of measuring character height. This illustration shows a lower case letter g inside the nominal character box. This box consists of the up vector (pointing north in this case), the base vector (pointing east), and the top line. The base line is located above the base vector. The cap line is located below the top line. The bottom of the lower case g (tail) touches the base vector. The bottom of the circular part of the g touches the base line. The character height (h) is defined as the distance between the cap line and the base line. The width (w) defines how wide the character box should be.

Annotation Height Scale Factor: The Set Annotation Height Scale Factor **(GPAHSC)** subroutine creates a structure element that controls the actual character height (h_a) of annotation text by multiplying the workstation's nominal annotation character height (h_n), by the annotation height scale factor (s). Your application program can obtain the nominal annotation height and the permitted range of values for this attribute from the Inquire Extended Annotation Font Characteristics **(GPQXAF)** subroutine.

The actual width (w_a) of an annotation character can be calculated as follows:

$w_a = h_a \times e \times ar_n$

where:

e is the character expansion factor

ar_n is the nominal character aspect ratio.

Both the character expansion factor and nominal aspect ratio are discussed further in the following paragraphs.

Character Expansion Factor: The Set Character Expansion Factor **(GPCHXP)** subroutine sets the actual character aspect ratio (ar_a) relative to the nominal aspect ratio (ar_n) for all subsequent geometric or annotation text primitives. The nominal aspect ratio is defined as follows:

$$ar_n = \frac{w_n}{h_n}$$

Where:

w_n is the nominal character width in font coordinates and

h_n is the nominal character height (capline to baseline) in font coordinates.

The actual character aspect ratio (actual width divided by actual height) is defined below:

$$ar_a = \frac{w_a}{h_a} = e \times ar_n$$

where:

w_a is the actual character width in font coordinates and

h_a is the actual character height (capline to base line) in font coordinates

e is the character expansion factor

So, the actual character width (w_a) is the product of the actual aspect ratio (ar_a) and the actual character height (h_a).

The nominal aspect ratio (ar_n) for geometric and annotation text characters can be obtained using the Inquire Font Characteristics **(GPQFCH)** and Inquire Extended Annotation Font Characteristics **(GPQXAF)** subroutines. The aspect ratios of proportionally displayed geometric text can be obtained from the Inquire Font Aspect Ratios **(GPQFAR)** subroutine. Whether or not a font is proportional can be determined from **GPQFCH** or **GPQXAF**, and workstation support for proportional fonts can be inquired using the Inquire Extended Text Facilities **(GPQXTX)** subroutine. Also, the Inquire Geometric Font Characteristics **(GPQGFC)** subroutine returns the range of expansion factors that a workstation supports.

Character Up Vector: The Set Character Up Vector **(GPCHUP)** subroutine creates a structure element which defines the character up vector for geometric text primitives with respect to the text reference coordinate system. The character up vector attribute can be used to rotate geometric text within the text reference plane as shown in the following figure:

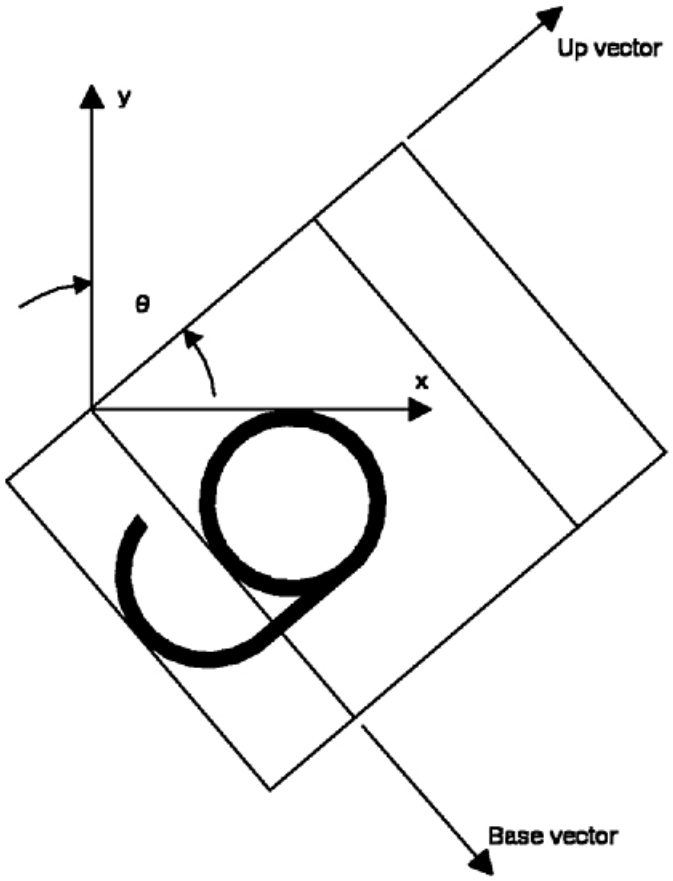


Figure 60. Rotating a character using the character up vector. This illustration shows the same character box as described in the previous figure. In this case, the up vector is pointing in the northeast direction, and the base vector is pointing in the southeast direction. This causes the lower case g and the character box to appear angled.

The character base vector is equal to the character up vector rotated 90 degrees clockwise in the text reference plane. If you wish to also specify the character base vector then your application should call the Set Character Up and Base Vector **(GPCHUB)** subroutine defined in the advanced attributes section.

Character Spacing: The Set Character Spacing **(GPCHSP)** subroutine creates a structure element that sets the amount of space between character boxes for both geometric and annotation text primitives. The character spacing is specified as a fraction of the nominal character height (h_n) so the actual space is equal to the following:

$s_a = s_f \times h_n$

where:

s_a is the actual character spacing

s_f is the fractional character spacing set using **GPCHSP**

h_n is the nominal character height.

The actual height ($_a$) for geometric text is set using the Set Character Height **(GPCHH)** subroutine. The actual height (h_a) for annotation text is set with either the Set Annotation Height Scale Factor **(GPAHSC)** subroutine or the Set Annotation Height **(GPAH)** subroutines.

Text Color Index: The Set Text Color Index **(GPTXCI)** subroutine creates a structure element which specifies the color of geometric text as an index into the workstation's rendering color table.

If the font is defined as ″filled″, each character will be solid filled using the Text Color attribute.

Text Font: The Set Text Font **(GPTXFO)** subroutine creates a structure element which sets the font in which subsequent text primitives will be rendered. The actual font used to render the text primitives depends upon the current character set defined using the Set Character Set **(GPTXCS)** subroutine. **GPTXCS** does not create a structure element but specifies which character set will be used when a text primitive is defined. The Inquire Character Set Identifier **(GPQCS)** subroutine returns the current Text Character Set to your application program.

For more information concerning character sets and fonts see Chapter 4. Structure Elements and Chapter 19. Fonts.

Text Index: The Set Text Index **(GPTXI)** subroutine creates a structure element which is an index into the workstation's text bundle table. Depending upon the setting of the text Attribute Source Flags (ASFs), text primitives will be rendered with either bundled or individual attributes.

For more information on the use of bundled text attributes, see Chapter 4. Structure Elements.

Text Path: The Set Text Path **(GPTXPT)** subroutine creates a structure element which specifies the direction in which character boxes, containing successive characters from the text string in the primitive, are concatenated to obtain the text primitive's extent rectangle. The text path attribute is discussed in detail in Chapter 4. Structure Elements.

Advanced Geometric and Annotation Text Attributes

Geometric Text Culling: The Set Geometric Text Culling **(GPGTXC)** subroutine is an escape which defines the geometric text culling height, in Device Coordinates (DC), for all subsequent geometric text primitives being displayed on the workstation. If the height of transformed geometric text is less than the specified culling height, the text is displayed using one of the following text culling methods:

1=TEXT_DISPLAY displays each stroke of each character (default method)

2=BOX_DISPLAY draws a text-colored polyline around the text extant rectangle

3=NO_DISPLAY causes the character text not be displayed

The use of the Set Geometric Text Culling attribute can increase the performance of your application by reducing the amount of processing required to display text smaller than the specified culling height.

Character Positioning Mode: The Set Character Positioning Mode **(GPCHPM)** creates a structure element which controls whether or not characters are drawn with the same width character box (`1=CONSTANT`) or with the character box of each individual character (`2=PROPORTIONAL`). Each font has a width to be used for `CONSTANT` positioning mode. In addition to this, proportional fonts have individual widths defined for each character. Information on the calculation of actual character widths is in the discussion of the Set Character Expansion Factor attribute earlier in this chapter. A discussion on fonts is found in Chapter 19. Fonts.

Use the Inquire Extended Text Facilities **(GPQXTX)** subroutine to determine if a workstation supports proportionally spaced fonts.

Character Up and Base Vectors: The Set Character Up and Base Vectors **(GPCHUB)** subroutine creates a structure element which rotates and shears the text extent rectangle in the local font coordinate system as shown in the following figure:

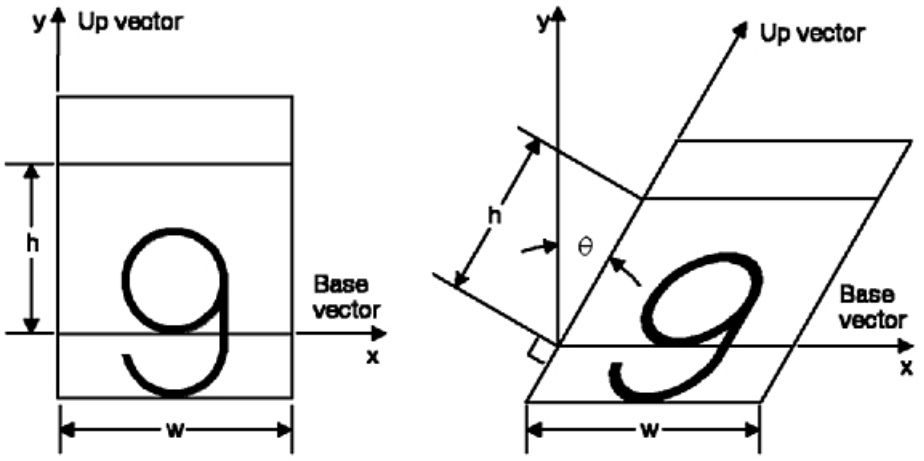


Figure 61. Shearing text using the character up and base vectors. This illustration shows two views of the character box containing the lower case letter g. The first view is the same as the figure above entitled Example of measuring character height. The second view shows the up vector rotated clockwise (in the northeast direction) by a value of theta. The base vector, character height (h), and width (w) remain unchanged. Hence, the character box and letter appear rotated and sheared.

The base vector defines the rotation of the text string and the up vector defines the shear of the text extent rectangle. The Set Character Up and Base Vectors attribute is an extended form of the Set Character Up Vector attribute which explicitly sets the up vector and implicitly sets the base vector to be perpendicular to the up vector.

When using the Character Up and Base Vector attribute to shear text you should note that the character height is the length of the edge of the character box along the character up vector from the baseline to the capline. The height is *not* the perpendicular distance between the capline and baseline unless the angle between the up and base vectors is 90 degrees (i.e. no shear). The width of a character is always the length of the character extent box along the character base vector.

The text parallelogram is constructed by concatenating rotated and sheared character boxes side to side or top to bottom according to the text path attribute. The parallelogram is aligned with respect to the character up and base vectors as shown in the following figure:

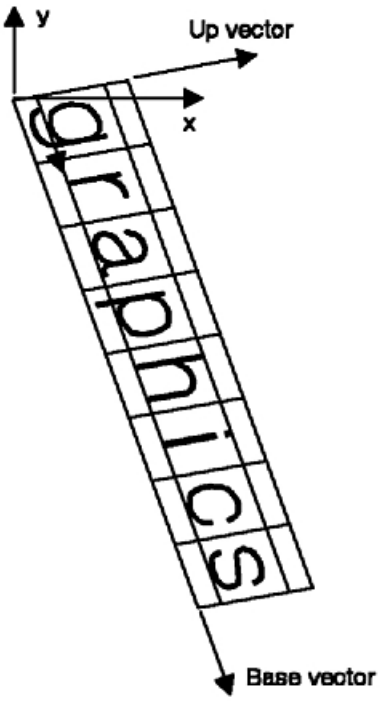


Figure 62. Rotating and shearing the text extent rectangle. This illustration depicts a series of letters in character boxes concatenated to form a parallelogram The series of letters form the word graphics. In this case, the up vector is pointing in a northeastern direction (rotated about 80 degrees from north), and the base vector points in a southeastern direction (rotated about 160 degrees from north). This causes the text to appear angled and sheared in a diagonal direction.

Text Color Direct: The Set Text Color Direct **(GPTXCD)** subroutine creates a structure element which specifies the color of geometric and annotation text as a color vector. For more information on colors and how they are specified and processed, see Chapter 17. Manipulating Color and Frame Buffers.

Annotation Alignment: The Set Annotation Alignment **(GPAAL)** subroutine creates a structure element which defines the alignment point for all subsequent annotation text primitives relatives to the primitive's extent rectangle.

Annotation Height: The Set Annotation Height **(GPAH)** subroutine creates a structure element which specifies annotation character height in Normalized Projection Coordinates (NPC). The annotation height attribute is equivalent to the Annotation Height Scale Factor attribute and yields the same annotation character height in Device Coordinates (DC). At structure traversal time the annotation character height, in DC, is obtained by multiplying the Annotation Height by the scale of the workstation transformation. An equivalent Annotation Scale Factor is then obtained by dividing the annotation character height in DC by the nominal annotation character height in DC. Your application can get the nominal height and permitted annotation height expansion factors using the Inquire Extended Annotation Font Characteristics **(GPQXAF)** subroutine. The locations of the capline and baseline in the character box for the nominal height are also returned.

Annotation Path: The Set Annotation Path **(GPAPT)** subroutine creates a structure element which specifies the direction that successive characters from a text primitive are concatenated to obtain the text extent rectangle. The Annotation Path attribute is similar to the Text Path attribute applied to geometric text which is described fully in Chapter 4. Structure Elements.

Annotation Style: The Set Annotation Style **(GPAS)** subroutine creates a structure element that specifies the drawing style used for subsequent Annotation Text Relative **(GPANR3**/**GPANR2)** primitives. The Annotation Text Relative primitives specify a reference point and an offset from the point of where the text is drawn. The Set Annotation Style attribute specifies a method for connecting the reference point to the

point specified by the offset parameter. By using the 2=LEAD_LINE style, the two points are connected by a polyline drawn using the current polyline attributes. A style of 1=UNCONNECTED can also be specified. When the reference point specified in the Annotation Text Relative primitive is clipped, the text and any connecting line are not displayed. If the reference point is not clipped, then the text and connecting line are displayed and are subject to the normal clipping rules.

Annotation Up Vector: The Set Annotation Up Vector **(GPAUP)** subroutine creates a structure element which rotates annotation text within the text reference plane by setting the annotation character up vector with respect to the text reference coordinate system. The character base vector is set to the character up vector rotated 90 degrees clockwise in the text reference plane. The effect of this attribute is similar to the Character Up Vector attribute applied to geometric text which is illustrated in the following figure:

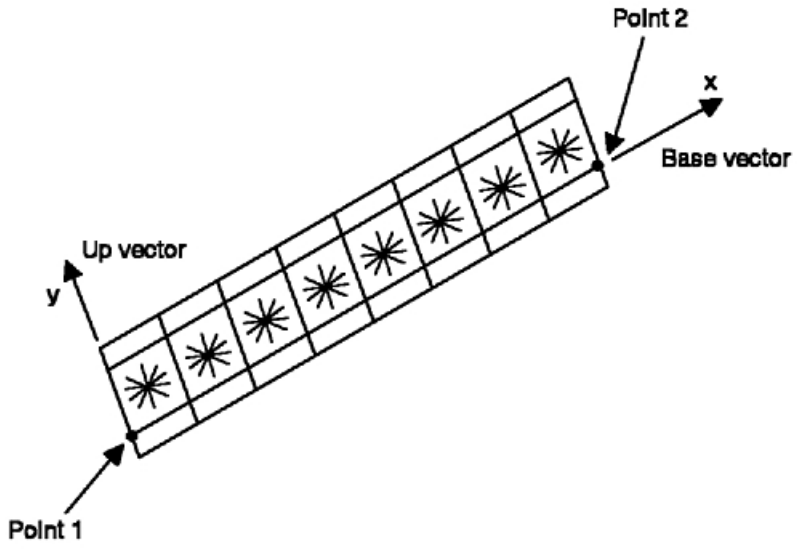


Figure 63. Font coordinate system for character line primitives. Font coordinate system for character line primitives. This illustration shows a series of character boxes strung together to form a rectangle. Each character box contains an asterisk. The up vector is pointing in a northeastern direction (rotated about -20 degrees from north), and the base vector is pointing in the northwestern direction (rotated 90 degrees from the up vector). The characters, along with the character boxes, are pictured at an angle going upward.

Bundle Tables

Each workstation has a text attribute bundle table which contains multiple entries that each define a set of text attributes. An entry in a text attribute bundle table can be set by Set Extended Text Representation **(GPXTXR)** subroutine. To select an entry in a text attribute bundle table your application can create a Set Text Index structure element which selects a text bundle table entry during structure traversal. A Set Text Index element is generated using the Set Text Index **(GPTXI)** subroutine. The attributes in the selected bundle table entry will only take effect if the appropriate text Attribute Source Flags (ASFs) are set to 1=BUNDLED using the Set Attribute Source Flags **(GPASF)** subroutine.

The number of entries in a text attribute bundle table can be inquired using the Inquire Extended Text Facilities **(GPQXTX)** subroutine. Also, the Inquire Predefined Text Representation **(GPQPTR)** subroutine returns the predefined values for text bundle table entries. The Inquire Extended Text Representation **(GPQXTR)** subroutine returns the current text bundle table contents.

The following text attributes can be specified as either 1=BUNDLED or 2=INDIVIDUAL:

- Character expansion factor
- Character spacing
- Text color
- Text font
- Text precision.

For more information on the use of bundled text attributes, please see Chapter 4. Structure Elements.

Character Line Attributes

Some of the attributes applied to geometric text are also applied to the Character Line 2 primitive.

Due to the definition of the character line primitive, the following geometric text attributes do not apply to the primitive or are derived directly from the primitive's definition:

- Character expansion factor
- Character up and base vectors
- Character height
- Character spacing
- Text alignment
- Text path
- Text precision

The following attribute descriptions list each text attribute and how the attribute is actually applied to character line primitives.

Character Line Scale Factor: The Set Character Line Scale Factor **(GPCHLS)** subroutine creates a structure element which controls the actual character height of all subsequent character line primitives. The actual character height is the product of the nominal character height and the character line scale factor. The nominal character height is supplied as a parameter when the character line primitive is created.

Character Positioning Mode: Fonts used by the graPHIGS API can be defined as either proportional of fixed. Using the character positioning mode attribute, the use of the proportional information can be either turned on or off. Proportionally defined fonts will be displayed as fixed spaced fonts when the character positioning mode is set to 1=CONSTANT using the Set Character Positioning Mode **(GPCHPM)** subroutine. When the character positioning mode is set to 2=PROPORTIONAL as opposed to 1=CONSTANT, the repeated characters of a character line primitive will generally be more tightly spaced.

Character Expansion Factor: The character expansion factor attribute is ignored for character line primitives. Instead, a character line primitive is drawn in an integral number of character boxes between the end points of the character line. The width of the character box is the nominal width scaled by the Character Line Scale Factor. The nominal width is determined by the Character Positioning Mode and the font definition. This width is compressed as little as possible to fit in an exact number of characters.

Character Height: The actual height (h_a) is the product of the character height contained in the character line definition and the character line scale factor (s).

Character Spacing: The character spacing attribute is ignored for character line primitives. A spacing of zero is used instead.

Text Alignment: By definition, a character line primitive is always aligned at the left base of the primitive's text extent rectangle.

Text Path: The text path attribute is ignored for character line primitives.

Text Precision: Character line primitives are always rendered with STROKE precision.

Text Color Index: The Set Text Color Index **(GPTXCI)** subroutine creates a structure element which controls the color of subsequent character line primitives. The specified color is an index into the workstation's rendering color table.

Text Font: The Set Text Font **(GPTXFO)** subroutine creates a structure element which sets the font in which subsequent character line primitives are rendered. The font is taken from the current character set which your application can change using the Set Text Character Set **(GPTXCS)** subroutine. The Inquire Character Set Identifier **(GPQCS)** subroutine returns the current Text Character Set to your application program.

Text Index: The Set Text Index **(GPTXI)** subroutine creates a structure element which is an index into the workstation's text bundle table.

Character Up and Base Vectors: The Character Up and Base Vectors are always (0,1) and (1,0) in the local font coordinate system. As shown in the figure, Font coordinate system for character line primitives, the x-reference vector in the font coordinate systems lies along the line from the character line's starting point toward the end point. The y-reference vector is the x-reference rotated counterclockwise 90 degrees.

Text Color Direct: The Set Text Color Direct **(GPTXCD)** subroutine creates a structure element which sets the Text Color attribute for all subsequent character line primitives.

If the character used in the character line primitive is marked as filled in the font definition, the character will be solid filled using the current Text Color attribute.

Bundle Tables: The only text bundle table attributes which apply to character line primitives are the following:

- text color
- text font.

Polygon Attributes

Polygon attributes apply to the following primitives:

- Polygon 2/3
- Polygon with Data 2/3
- Composite Fill Area 2
- Triangle Strip 3
- Non-Uniform B-Spline Surface
- Trimmed Non-Uniform B-Spline Surface
- Polysphere.

Basic Polygon Attributes

The following basic polygon attributes are discussed in detail in Chapter 4. Structure Elements.

Interior Index: The Set Interior Index **(GPII)** subroutine creates a structure element which is an index into the workstation's interior bundle table. The index is used to select an entry in the interior bundle table which contains interior attributes such as interior style and color.

Interior Style: The Set Interior Style **(GPIS)** subroutine creates a structure element which sets the interior style of all subsequent area defining primitives. The style can be one of the following:

- 1=HOLLOW
- 2=SOLID
- 3=PATTERN

- 4=HATCH
- 5=EMPTY

Each of the interior styles listed above are discussed in Chapter 4. Structure Elements.

Interior Style Index: The Set Interior Style Index **(GPISI)** subroutine creates a structure element which is an index into the workstation's pattern or hatch table depending on the setting of the interior style attribute.

Interior Color Index: The Set Interior Color Index **(GPICI)** subroutine creates a structure element which sets the interior color by specifying an index in the workstation's rendering color table. For more information on color tables and color processing, see Chapter 17. Manipulating Color and Frame Buffers.

Edge Index: The Set Edge Index **(GPEI)** subroutine creates a structure element which is an index into the workstation's edge bundle table. Depending on the settings of the edge attribute source flags, the edge bundle table entry selected by the index will control the line type, scale factor, and color of polygon edges.

Edge Line Type: The Set Edge Line Type **(GPELT)** subroutine creates a structure element which sets the line type of polygon edges by specifying an index in the workstation's line type table.

Edge Scale Factor: The Set Edge Scale Factor **(GPESC)** subroutine creates a structure element which sets the edge scale factor for polygon edges. The edge scale factor specifies the width of a polygon edge as a fraction of the nominal edge width.

Edge Color Index: The Set Edge Color Index **(GPECI)** subroutine creates a structure element which sets the color of polygon edges by specifying an index in the workstation's rendering color table. For more information on color tables, see Chapter 17. Manipulating Color and Frame Buffers.

Attribute Source Flags: The Set Attribute Source Flag **(GPASF)** subroutine creates a structure element which sets the attribute source flag for the interior style, interior style index, interior color, edge flag, edge line type, edge scale factor, and edge color attributes. Each attribute source flag specifies that the attribute is defined either as an individual structure element or in an entry in a workstation bundle table. For more information on attribute source flags and the use of bundle tables, see Chapter 4. Structure Elements.

Advanced Polygon Attributes

Most of the advanced attributes affect all area defining primitives, including surfaces. However, some of the attributes, such as the Surface Approximation Criteria, affect surface primitives only.

An advanced property of all polygon primitives is that they are defined to be front or back facing. A polygon is front facing if its surface normal is pointing toward the viewer and is back facing if its surface normal is facing away from the viewer. The surface normals can be specified for some primitives, such as the Polygon with Data and Triangle Strip primitives, but for others, normals are calculated based on the primitive's definition. By default all polygons are treated as front facing.

To enable back face differentiation, your application must use the 2=COLOR_AND_SURFACE_PROPERTIES mode using the Set Face Distinguish Mode **(GPFDMO)** subroutine. Your application is given the ability with some of the subroutines presented in this section to assign different properties to the front and back faces of polygon primitives. The following polygon attributes are affected by the Face Distinguish mode:

- Interior Color
- Specular Color
- Surface Properties

Note that surface primitives, such as polyspheres, are actually rendered using polygons. Each polygon has an associated surface normal and is independently processed when front and back face attributes are applied.

Many of the polygon attributes in the following paragraphs are related to lighting and shading. For a full discussion of lights, specular color, and surface properties, refer to Chapter 16. Rendering Pipeline.

Interior Color Direct: The Set Interior Color Direct **(GPICD)** subroutine creates a structure element which sets the interior color for all polygon primitives. This attribute element is similar to the Set Interior Color Index **(GPICI)** subroutine except that the color is specified with a color vector instead of an index into the workstation's rendering color table. If the Face Distinguishing Mode is 2=COLOR_AND_SURFACE_PROPERTIES, then the interior color direct attribute only effects front facing polygons.

Edge Color Direct: The Set Edge Color Direct **(GPECD)** subroutine creates a structure element which sets the edge color for all polygon primitives. This attribute element is similar to the Set Edge Color Index **(GPECI)** subroutine except that the color is specified with a color vector instead of an index into the rendering color table.

Back Interior Color Index: The Set Back Interior Color Index **(GPBICI)** subroutine creates a structure element which sets the interior color for back facing polygons by specifying an index in the rendering color table. The back interior color index attribute only takes effect if the Face Distinguish Mode is set to 2=COLOR_AND_SURFACE_PROPERTIES.

Edge Flag: The Set Edge Flag **(GPEF)** subroutine creates a structure element which controls whether or not the edges of an area defining primitives are drawn. The available Edge Flag values are 1=OFF, 2=ON, and 3=GEOMETRY_ONLY When the Edge Flag is set to 3=GEOMETRY_ONLY, the graPHIGS API renders (for display) the polygon in the same manner as when the Edge Flag is 1=OFF of the polygon is written to the Z-buffer (used during HLHSR processing) using a line method rather than an area-fill method. This guarantees that a line drawn around the boundary of the polygon will be coincident with the boundary of the polygon preventing artifacts due to the difference in drawing algorithms.

Back Interior Color Direct: The Set Back Interior Color Direct **(GPBICD)** subroutine creates a structure element which sets the interior color for all back facing polygon primitives. This structure element is similar to Back Interior Color Index structure element except that the color is defined by a color vector instead of an index into a workstation's rendering color table. The back interior color direct attribute only takes effect if the Face Distinguishing Mode is set to 2=COLOR_AND_SURFACE_PROPERTIES.

Polygon Culling: The Set Polygon Culling **(GPPGC)** subroutine creates a structure element which sets the polygon culling mode. The available culling modes are: no polygons are culled (1=NONE), back facing polygons are culled (2=BACK), and front facing polygons are culled (3=FRONT). If a polygon is 'culled', it is taken out of the rendering pipeline and not displayed. This subroutine call may be used to increase performance, when displaying polyspheres for example. The Inquire Advanced Attribute Facilities **(GPQAAF)** subroutine can be used by your application to determine which polygon culling modes are supported.

Face Distinguish Mode: Set Face Distinguish Mode **(GPFDMO)** subroutine creates a structure element which sets the Face Distinguish Mode to 1=NONE or 2=COLOR_AND_SURFACE_PROPERTIES When the Face Distinguish Mode is 1=NONE, all polygons are treated like front facing polygons. When Face Distinguish Mode is set to 2=COLOR_AND_SURFACE_PROPERTIES, back facing polygons inherit back facing attributes only. The default mode is 1=NONE.

Light Source State: The Set Light Source State **(GPLSS)** subroutine creates a structure element which sets the light source state. The light source state is a list of active light source indexes, each of which points to an entry in the light source table. The Light Source State structure element has two lists of light source indexes; an activate and deactivate list. These two lists are applied to the light source state during traversal. The default light source state has no entries.

Lighting Calculation Mode: The Set Lighting Calculation Mode **(GPLMO)** subroutine creates a structure element which sets the Lighting Calculation Mode. Lighting calculations are done once per polygon (2=PER_AREA) on the facet normal (flat shading), or at each vertex (3=PER_VERTEX) of every polygon if the

data is available (Gouraud shading). The Inquire Advanced Attribute Facilities **(GPQAAF)** subroutine can be used by your application to determine which Lighting Calculation Modes are supported. The default Lighting Calculation Mode is 1=NONE.

Reflectance Model: The lighting process involves the calculation of color values from the light source values and the primitive attribute values, such as the surface properties and specular color. The Set Reflectance Model **(GPRMO)** and the Set Back Reflectance Model **(GPBRMO)** subroutines specify which terms of the calculation are performed. The calculation can include the ambient, diffuse, and specular terms (4=AMB_DIFF_SPEC), or the ambient and diffuse terms (3=AMB_DIFF), or the ambient term only (2=AMB). The reflectance calculations are conceptually performed at each position of the interior of the primitive. However, this calculation at every pixel may not be performed depending on the Interior Shading Method selected. (See Step Two: Reflectance Calculations and Shading for more information.)

Shading: The Set Interior Shading Method **(GPISM** and **GPBISM)** subroutines define how to fill the interior of area primitives. Except for 1=SHADING_NONE, the shading methods use interpolation to perform the fill process. Interpolation is the process of calculating data at intermediate points along a primitive's geometry from data defined at the primitive's vertices. It results in averaging the data across a primitive. (See Shading).

Specular Color Index: The Set Specular Color Index **(GPSCI)** subroutine creates a structure element which sets the specular color for all polygon primitives by specifying an index in the workstation's rendering color table. If the Face Distinguish Mode is 2=COLOR_AND_SURFACE_PROPERTIES, then the specular color attribute only effects front facing polygons.

Specular Color Direct: The Set Specular Color Direct **(GPSCD)** subroutine creates a structure element which sets the specular color for all polygon primitives. This attribute element is similar to Set Specular Color Index **(GPSCI)** except that the color is specified with a color vector instead of an index into the rendering color table. If the Face Distinguishing Mode is 2=COLOR_AND_SURFACE_PROPERTIES, then the specular color attribute only effects front facing polygons.

Back Specular Color Index: The Set Back Specular Color Index **(GPBSCI)** subroutine creates a structure element which sets the specular color for all back facing polygon primitives by specifying an index in the rendering color table. A back specular color attribute only takes effect if the Face Distinguish Mode is set to 2=COLOR_AND_SURFACE_PROPERTIES.

Back Specular Color Direct: The Set Back Specular Color Direct **(GPBSCD)** subroutine creates a structure element which sets the specular color for all back facing polygon primitives. This attribute element is similar to Back Specular Color Index attribute except that the color is defined as a color vector instead of an index into a workstation's rendering color table. As with the back specular color index attribute, this attribute takes effect only if the Face Distinguishing Mode is set to 2=COLOR_AND_SURFACE_PROPERTIES.

Surface Properties: The Set Surface Properties **(GPSPR)** subroutine creates a structure element which sets the surface properties for all polygon primitives. These surface properties are:

- Ambient Reflection Coefficient
- Diffuse Reflection Coefficient
- Specular Reflection Coefficient
- Specular Reflection Exponent
- Transparency Coefficient

The surface properties effect both front and back facing surfaces unless the Face Distinguishing Mode is set to 2=COLOR_AND_SURFACE_PROPERTIES.

Back Surface Properties: The Set Back Surface Properties **(GPBSPR)** subroutine creates a structure element which sets the surface properties for all back facing polygon primitives but only if the Face Distinguish Mode is set to 2=COLOR_AND_SURFACE_PROPERTIES.

Data Mapping (Texture Mapping): Data mapping uses data values from the vertices of primitives to determine the colors to be used to render the primitives. (See Color Selection for a description of the color selection process and how it relates to data mapping.) The Set Data Mapping Representation **(GPDMR)** subroutine provides the values that specify how the data mapping entry is accessed from the data values in the primitive, and how the data mapping entry is organized. The Set Data Mapping Index **(GPDMI)** or Set Back Data Mapping Index **(GPBDMI)** specifies the entry in the data mapping representation table used to perform the mapping. Use the Inquire Workstation Description **(GPQWDT)** subroutine to determine whether data mapping is supported on your workstation. Refer to Texture/Data Mapping for information about using data mapping to implement *texture mapping*, and an explanation of the overall data mapping process.

Data Morphing and Vertex Morphing: Morphing modifies the geometry and/or data (*texture*) mapping data of a primitive. There are two types of morphing, one using vertex coordinates and the other using data mapping data values, called *vertex morphing* and *data morphing*, respectively. Vertex morphing allows you to modify the rendered geometry of the primitive without changing the structure element. You define the changes using morphing values at each vertex of the affected primitives along with scale factors that define how the vertex morphing values affect the primitive. Use the Set Vertex Morphing Factors **(GPVMF)** subroutine to specify the scale factors. You can only specify Data morphing values if corresponding data mapping data values exist in the primitive definition. Data morphing occurs in data space before all transformations are applied to the data values. Data morphing changes the vertex data mapping values, before the data matrix is applied. You use the Set Data Morphing Factors **(GPDMF** and **GPBDMF)** subroutines to specify the data morphing scale factors. Use the Inquire Workstation Description **(GPQWDT)** call to determine whether your workstation supports the graPHIGS API morphing facilities. For more detailed information, see Morphing.

Modeling Clipping: The modeling clipping function provides a way to clip geometric entities in world coordinates. The clipping volume in world coordinates is set by the Set Modeling Clipping Volume 2/3 **(GPMCV2)** and **(GPMCV3)** elements. Modeling clipping is activated or deactivated by the Set Modeling Clipping Indicator **(GPMCI)** element. Use the Inquire Workstation Description **(GPQWDT)** subroutine to determine whether modeling clipping is supported on your workstation. See Modeling Clipping for more information.

Transparency (Alpha Blending): You can assign an amount of transparency to objects by specifying the value of the transparency coefficient parameter (0.0, 1.0) using the Set Transparency Coefficient **(GPTCO)**, the Set Back Transparency Coefficient **(GPBTCO)** or the Set Surface Properties **(GPSPR)** subroutines. The source and destination blending functions specify the methods used to blend the source and destination colors. These are specified using the Set Blending Function **(GPBLF)** and Set Back Blending Function **(GPBBLF)** subroutines. The 3=BLEND processing mode of the Set Extended View Representation **(GPXVR)** subroutine blends area primitives such as polygons; to blend all primitives, use 4=BLEND_ALL Use **GPQTMO** to inquire this and other transparency mode facilities for your workstation, and **GPQWDT** to inquire the number and availability of source and destination blending functions on your workstation. For more detailed information, see Transparency.

Parametric Surface Characteristics: The Set Parametric Surface Characteristics **(GPPSC)** subroutine creates a structure element which defines the type of line geometry used to render the interior of parametric surfaces. When your application specifies 1=NONE, no line geometry is generated for subsequent parametric surfaces. 1=NONE is the default if the Set Parametric Surface Characteristics attribute is not specified. When your application specifies 2=ISOPARAMETRIC_LINES, lines of constant *u* and *v* are drawn in the *u* and *v* parameter space of the surface. To control the spacing of the lines, your application must also supply the following information:

- The number of isoparametric lines in the *u* direction
- The number of isoparametric lines in the *v* direction

- The scope of the two isoparametric numbers indicating whether the numbers refer to the entire surface (SURFACE scope) or the patches between surface knots (BETWEEN_KNOTS scope).

When the scope is set to SURFACE, the specified number of isoparametric lines are drawn between the minimum and maximum parameter limits of the surface. A value of one results in two isoparametric lines, one in the *u* direction and one in the *v* direction at $(v_{min} + v_{max})/2$ and $(u_{min} + u_{max})/2$ respectively. When the scope is set to BETWEEN_KNOTS, the specified number isoparametric lines are drawn between each pair of knots in addition to the isoparametric lines. drawn at each knot value. So, a value of zero results in an isoparametric line drawn at each knot value within the parameter limits of the surface, and a value of one results in an additional isoparametric line midway between each knot pair.

For trimmed surfaces, isoparametric lines are only displayed in the active region of the trimmed surface. This means that the isoparametric lines are clipped by the trimming curves.

Since isoparametric lines are line geometry, polyline attributes are applied when the isoparametrics are displayed.

The echoing of a surface by the logical PICK device consists of the edges of the surface in addition to the following:

- The tessellation lines when the Set Parametric Surface Characteristics type is set to 1=NONE
- The isoparametric lines when the Set Parametric Surface Characteristics type is set to 2=ISOPARAMETRIC_LINES

Your application can combine the use of interior styles, edge attributes, and parametric surface characteristics. When drawn, the edges of a surface have priority over the ISOPARAMETRIC_LINES, which have priority over the area rendering of the surface interior.

Surface Approximation Criteria: The Set Surface Approximation Criteria **(GPSAC)** subroutine creates a structure element which sets the surface approximation criteria for Non-Uniform Rational B-Spline (NURBS) Surfaces, Trimmed NURBS Surfaces, and Polyspheres. The Set Surface Approximation Criteria structure element controls the process of dividing each surface into pieces which can be drawn as flat polygons. This process, called ″tessellation″ is accomplished by dividing each span of each surface parameter into intervals. The following figure illustrates the results of tessellating a NURBS surface into a coarse polygon grid:

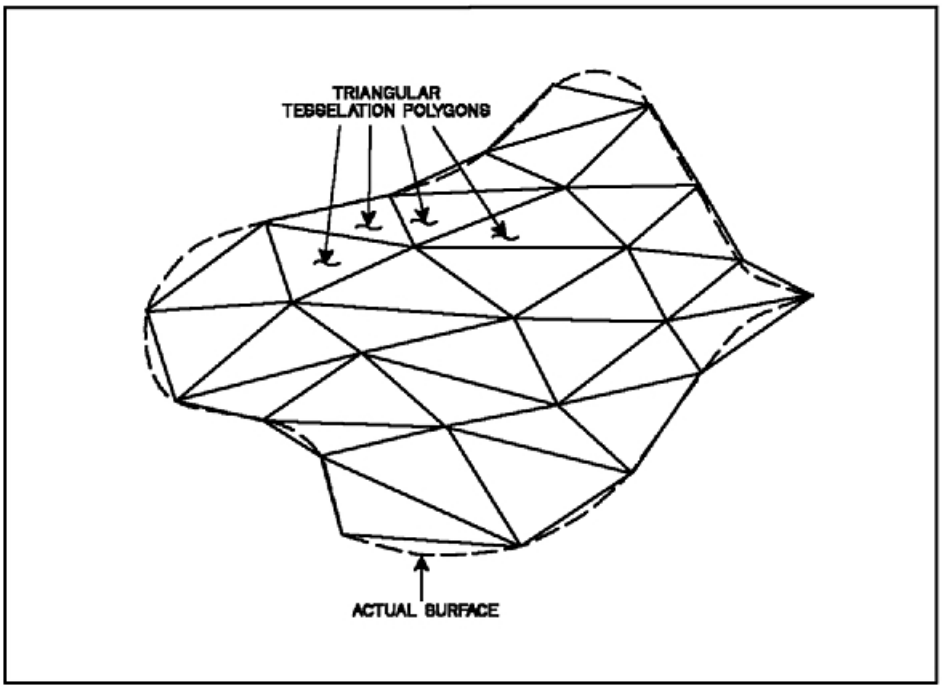


Figure 64. Tessellation of a NURBS surface into triangles. This illustration shows a surface divided into many flat, triangular polygons to approximate the surface. The polygons are called triangular tessellation polygons in the illustration.

The number of intervals used to represent each span is determined by the criterion and control values as summarized below:

(Criterion 1) `WORKSTATION_DEPENDENT`: This criterion relies on control values Q_u and Q_v which represent the quality of the surface approximation where 1.0 is nominal quality. A value of Q_u or Q_v greater than or less than 1.0 produces a higher or lower quality surface respectively. Q_u and Q_v are used to determine the number of tessellation intervals in the u (n_u) and v (n_v) directions using the following formulas:

$n_u = Q_u \times d_{ui} \times \text{sqrt}(s)$
$n_v = Q_v \times d_{vi} \times \text{sqrt}(s)$

where:

d_{ui} and d_{vi} are values determined by taking the chordal deviation for each interval.

s represents the current scale or ratio of world coordinates to screen coordinates.

(Criterion 3) `CONSTANT_SUBDIVISION_BETWEEN_KNOTS`: The simplest Surface Approximation Criterion involves specifying the number of subdivisions between knots in both the u and v directions as N_u and N_v respectively.

(Criterion 8) `VARIABLE_SUBDIVISION_BETWEEN_KNOTS`: In this case, the number of tessellation intervals in the u and v directions is equal to the following:

$n_u = Q_u \times f_{ui}$
$n_v = Q_v \times f_{vi}$

where:

Q_u and Q_v are the tessellation control values

f_{ui} and f_{vi} are the i^{th} tessellation data values supplied with the definition of the NURBS surface.

Note: Only criterion 1 applies to the polysphere primitive.

Trimming Curve Approximation Criteria: The Set Trimming Curve Approximation Criteria **(GPTCAC)** subroutine creates a structure element which sets the trimming curve approximation criteria applied only to Trimmed Non-Uniform B-Spline (NURBS) Surfaces. The trimming curve approximation criteria structure element controls the representation of trimming curves. Two of the parameters to **GPTCAC** are Q_{tu} and Q_{tv} which control the resolution of the surface near the trimming curves and are independent of the trimming curve approximation criterion. Large values of Q_{tu} and Q_{tv}(> 1.0) produce smooth surface edges near the trimming curves. Small values (< 1.0) produce jagged edges, but can improve the performance of Trimmed NURBS surfaces. The number of intervals used to represent each span of the trimming curve is determined by a criterion and the control value as defined below:

(Criterion 1) `WORKSTATION_DEPENDENT`: The workstation-dependent trimming curve approximation criterion relies on the control value Q_t that represents the quality of the trimming curve approximation where 1.0 is nominal quality. Values greater than or less than 1.0 produce a higher or lower quality trimming curves respectively. Q_t for criterion 1 is used as follows to calculate the number of tessellation intervals in used to render the trimming curve:

$n_t = Q_t \times d_{ti} \times \text{sqrt}\ (s)$

where:

d_{ti} is a value determined by taking the chordal deviation of the trimming curve for each interval.

s represents the current scale or ratio of world coordinates to screen coordinates.

(Criterion 3) `3=CONSTANT_SUBDIVISION_BETWEEN_KNOTS`: As with regular NURBS curve primitives, the simplest surface approximation criterion involves specifying a the number of subdivisions between knots as a constant

(Criterion 8) `8=VARIABLE_SUBDIVISION_BETWEEN_KNOTS`: In this case, the number of tessellation intervals is equal to the following:

$n_t = Q_t \times f_{ui}$

where:

Q_t is the tessellation control value

f_{ti} is the i^{th} tessellation data value supplied with the definition of the trimming curve.

Bundle Tables

In addition to the individual attributes discussed previously, polygon attributes can also controlled by bundle tables. The Set Extended Edge Representation **(GPXER)** subroutine is an advanced version of the Set Edge Representation **(GPER)** subroutine. Both allow your application to define bundle table entries for polygon edge attributes. **GPXER** extends the capabilities of **GPER** by allowing your application to define polygon edge colors directly as well as with a color index.

Set Extended Interior Representation **(GPXIR)** is the advanced version of the Set Interior Representation **(GPIR)** subroutine. Both subroutines allow your application to define bundle table entries for polygon

interior attributes. Like **GPXER**, **GPXIR** allows your application to define polygon interior colors directly as well as with a color index. To determine the number of entries in a workstation's bundle table, your application may use the Inquire Actual Length of Workstation State Tables **(GPQALW)** subroutine.

Application Defined Hatch Patterns

The Set Hatch Representation **(GPHR)** subroutine allows your application to define a hatch pattern entry in a workstation's hatch table. The hatch entry is specified via a data format supported by the workstation. Currently, only one format, the `BIT ARRAY` is defined. With this format, each hatch is defined by a two-dimensional array of bits. Similar to the pattern interior style, this array is repeated in both x and y directions and mapped to the interior of the polygon primitive. The pixels which have 1s at the corresponding positions are filled with the current interior color.

To determine the number of entries in the hatch table, your application can use the Inquire Hatch Facilities **(GPQHF)** subroutine. The current contents of the hatch table entry can be determined by using the Inquire Hatch Representation **(GPQHR)** subroutine.

Chapter 12. Structure Concepts

This chapter describes advanced functions that control the general organization of graphical data.

Conditional Structure Execution

The conditional structure execution capabilities of the graPHIGS API allow your application to create conditional executes and returns in a structure. These facilities can be used to increase performance by eliminating unnecessary traversal and provide your application with a way to control traversal.

Conditional executes and returns are based on tests performed against a *condition flag* The condition flag is a 32-bit string where bits 0 to 15 are reserved for use by the graPHIGS API application, and bits 16 to 31 are reserved for use by the graPHIGS API The traversal default of the condition flag is all 0s.

In order to set the bits in the condition flag, three condition setting structure elements have been defined. First, the Test Extent 2 **(GPTEX2)** and the Test Extent 3 **(GPTEX3)** subroutines create structure elements which set bits 30 and 31 of the condition flag. Bit 30 is called the *cull flag*, and bit 31 is called the *prune flag* The parameters to **GPTEX2** and **GPTEX3** are defined as follows:

extent Two diagonal points of an extent box in Modeling Coordinates (MC). The edges of the box are parallel to the axes of the MC system.

cull size index
An index specifying an entry in the workstation dependent cull size table. Each entry of the table is set by the Set Cull Size Representation **(GPCSR)** subroutine and includes a culling threshold in Device Coordinates (DC).

During structure traversal, the following actions are taken:

1. The specified extent box is transformed by the current modeling and viewing transformation and z-clipping applied based on the view specification. The modeling clip of the window and workstation clipping are not applied.
2. The smallest box whose edges are parallel to the Normalized Projection Coordinate (NPC) axes and which surrounds the remaining extent box after z-clipping is found.
3. If the box is completely outside the viewport of the view or completely outside of the workstation window, bit 31 (prune flag) is set to 1; otherwise it is set to 0.
4. The box is transformed by the workstation transformation and mapped onto 2D-DC by parallel projection.
5. If the diagonal of the mapped rectangle is shorter than the cull size specified by the cull size index, then bit 30 (cull flag) is set to 1; otherwise it is set to 0.

Through the Set Condition structure element, your application can directly set any condition flag. The element consists of two 32-bit bit strings: one on-flag and one off-flag. The bits of the current condition flag specified in the on-flag and off-flag are set to 1 and 0, respectively. More precisely, the resulting condition bits after this structure element is traversed becomes:

(current-flag OR on-flag) AND (NOT off-flag)

Two conditional structure execution elements can be created using the Conditional Execute Structure **(GPCEXS)** and Conditional Return **(GPCRET)** subroutines. The element created by the **GPCEXS** subroutine provides the following construct:

if the specified condition is satisfied **then** execute the specified structure **else** do nothing

The **GPCRET** subroutine provides the following construct:

if the specified condition is satisfied **then** terminate execution of this structure **else** continue execution of this structure

For both elements, the conditions are specified as follows:

flag mask
a 32-bit bit string specifying which bits in the condition flag will be tested.

criteria
an integer specifying how the condition flag bits will be tested.

(1: all 1s, 2: all 0s, 3: not all 1s, 4: not all 0s)

When the bits specified by the flag mask satisfy the specified condition, the operation in the **then** clause is executed, otherwise the **else** clause is executed.

Structure Manipulation

This section contains a discussion of structure manipulation in terms of editing modes, element pointer manipulation, element deletion, and structure inquiry.

Edit Mode

Your application program can edit structures using one of two editing modes - the Insert edit mode or the Replace edit mode. Using the INSERT_MODE, structure elements are inserted to the current open structure immediately after the element pointed to by the current element pointer. Then the current element pointer is moved to the element that was inserted.

In REPLACE_MODE, structure elements are not inserted, but replace the current element. If the current element pointer points to element 0, then the new element is inserted immediately before element 1. In either case, the current element pointer is set to point to the new element.

You may select either of the two edit modes by using the Set Edit Mode **(GPEDMO)** subroutine. When you issue the Open graPHIGS **(GPOPPH)** subroutine, the editing mode defaults to INSERT_MODE

Using the REPLACE_MODE is equivalent to calling the Delete Element **(GPDLE)** subroutine, followed by a subroutine call to an element generation function in INSERT_MODE For example, to replace a Polyline in REPLACE_MODE is equivalent to the following calling sequence in the INSERT_MODE:

```
CALL DELETE ELEMENT (GPDLE)
  CALL POLYLINE (GPPL2)
```

As another example, if you want to replace elements 8 through 10 with polylines, then use the following calling sequence:

```
  CALL SET EDIT MODE (INSERT_MODE)
  CALL DELETE ELEMENT RANGE (8,10)
  CALL POLYLINE (XXXX)
  CALL POLYLINE (XXXX)
  CALL POLYLINE (XXXX)
```

or the calling sequence:

```
  CALL SET EDIT MODE (REPLACE_MODE)
  CALL SET ELEMENT POINTER (8)
  CALL POLYLINE (XXXX)
```

```
CALL OFFSET ELEMENT POINTER (1)
CALL POLYLINE (XXXX)
CALL OFFSET ELEMENT POINTER (1)
CALL POLYLINE (XXXX)
```

Element Pointer Manipulation

The following element pointer manipulation subroutines are supported by the graPHIGS API:

- Set Element Pointer **(GPEP)**
- Offset Element Pointer **(GPOEP)**
- Set Element Pointer at Label **(GPEPLB)**
- Set Element Pointer at Pick Identifier **(GPEPPK)**
- Locate Element Pointer at Element Code **(GPEPCD)**
- Generalized Set Element Pointer at Label **(GPEPLG)**
- Generalized Set Element Pointer at Pick Identifier **(GPEPPG)**

The Locate Element Pointer at Element Code **(GPEPCD)** subroutine enables your application to locate the element pointer at the next occurrence of an element with any element code specified by your application. **GPEPCD** searches for the first occurrence of an element with the specified code starting at the current element. If the element is found before reaching the end of the structure, the current element pointer is set to point to the element. If no element with the specified element code has been found before reaching the end of the structure, an error is generated and the current element pointer is left unchanged.

Use the Generalized Set Element Pointer subroutines, (Generalized Set Element Pointer At Label **(GPEPLG)** and Generalized Set Element Pointer At Pick Identifier **(GPEPPG)**), to specify whether or not the search operation should wrap to the beginning of the structure if the label or pick identifier element is not found before the end of the structure is reached.

Element Deletion

The following element deletion subroutines are supported by the graPHIGS API:

- Delete Element **(GPDLE)**
- Delete Element Range **(GPDLER)**
- Delete Element Between Labels **(GPDELB)**
- Delete Element Group **(GPDLEG)**

The Delete Element Group **(GPDLEG)** subroutine will delete all the elements between the two specified label identifier elements and, depending on the option set by the application, may also delete the first label element and/or the second label element. The option also specifies how the subroutine searches for the labels (i.e., whether the search starts at the current element or whether it starts at the element following the current element).

After the element deletions, the element pointer moves to the element immediately preceding the deleted elements. If the search reaches the end of the structure without finding both of the label identifier elements, an error is generated and the current element pointer is left unchanged.

Structure Inquiry and Searching

Because structure contents are maintained by the graPHIGS API nucleus, each time the application issues a structure inquire subroutine call, communication between the shell and nucleus is required. When the nucleus is at another location (a remote nucleus), structure inquire subroutine calls may require a relatively large amount of I/O involving one round trip communication between the graPHIGS API shell and nucleus. Frequent use of these inquire functions is therefore not a good technique when using a remote nucleus.

Inquire List of Element Headers

The Inquire List of Element Headers **(GPQEHD)** subroutine, has been introduced as a way of minimizing I/O overhead. This subroutine has the following parameters:

- Number of entries requested
- Error indicator (return value)
- Number of entries actually returned (return value)
- A list of element headers (return value)

The element header returned by this subroutine has the following data format:

```
----------------------
| length  |  code    |
----------------------
```

The length field contains a 16-bit unsigned integer representing the length of the structure element including the header itself and the code field containing a 16-bit unsigned integer representing the structure element type.

The subroutine Inquire List Of Element Headers For Any Structure **(GPQEHA)** is a generalized form of **GPQEHD.** Using **GPQEHA**, your application can specify the structure identifier from which the data is obtained and a starting position in the structure from where to obtain the headers.

Note that the **GPQEHD** and **GPQEHA** subroutines return the element information in a different format and with different values than the Inquire Element Type And Size **(GPQETS)** subroutine. Although the latter subroutine is still supported, **GPQEHD** and **GPQEHA** are recommended for obtaining element headers from structures. Also, if your application is obtaining element headers in order to search for a particular element type, then you may be able to search more efficiently using the Element Search subroutine discussed in the Element Pointer Manipulation section on Element Pointer Manipulation. Although the latter subroutine is still supported, **GPQEHD** is recommended when searching a structure for a particular element.

Inquire List of Element Data

Another subroutine, Inquire List of Element Data **(GPQED)** subroutine, enables your application to inquire the content of one or more structure elements. The data returned by **GPQED** for each structure element is in the same order as the element headers returned by the Inquire List of Element Headers. **GPQED** requires the following parameters:

- The number of elements to be returned.
- The size of the buffer that the structure elements are to be placed in.
- An error indicator for returning any errors that are detected.
- An output parameter to receive the total number of elements that are returned. (This may be less than the number requested.)
- An output parameter to receive the sum of the lengths of all the elements that are returned.
- The buffer in which the elements are to be placed.
- An output parameter to receive the reason why the list of elements was terminated.

The termination reason parameter may indicate one of the following:

- The requested number of elements were returned. This is the usual case. The actual number returned equals the requested number.
- All the elements would not fit in the buffer provided by your application. The actual number indicates how many have been returned and may be zero.
- The end of the structure was encountered. The actual number indicates how many elements have been returned which may be equal to the requested number.
- One of the elements would not fit into the buffer between the shell and nucleus. The actual number indicates how many elements have been returned and that precede the one that would not fit.

The end of structure indication is important when the number returned equals the number requested. If your application tries to increment the element pointer by the number returned, the element pointer will be left at the last element of the structure since it cannot point beyond the end of the structure. If the application then issues another call to this subroutine, the last element that was returned on the previous subroutine call will be returned again. Your application may not be able to determine from the data content that duplicate elements have been returned. Therefore, your application should check the termination condition before offsetting the element pointer.

The elements are packed into the buffer provided by the application. The length in the header of each element will indicate the offset to the next structure element in the buffer.

Inquire List of Element Data for Any Structure

The subroutine, Inquire List Of Element Data For Any Structure **(GPQEDA)**, is a more generalized form of **GPQED.** Using **GPQEDA,** your application can specify the structure identifier from which the data is obtained, and a starting position in the structure from where to obtain the element data.

Inquire Ancestors/Descendants of Structure

Your application can use two inquiry subroutines to determine the structure relations created using Execute Structure subroutine. These relationships are returned as paths, which are lists of ordered pairs of information. Each element of the pair consists of:

- A structure identifier

  The identifier of a structure in the hierarchy related to the specified structure identifier
- An element number

  The element number of an EXECUTE STRUCTURE element in the structure

For example, if a path is returned as:

```
(101,10),(102,5),(103,0)
```

then the path is interpreted as follows:

- In structure 101, element 10 is an EXECUTE STRUCTURE of structure 102
- In structure 102, element 5 is an EXECUTE STRUCTURE of structure 103
- Structure 103 is the last structure of the path.

The returned data defines the path from structure 101 to structure 103. You can control the order of the path returned, so the top of the path is returned first or the bottom of the path is returned first. In addition, you can control the depth of the path returned (that is, the number of pairs of data returned).

The Inquire Ancestors Of Structure **(GPQPAS)** subroutine returns the paths that lead to the specified structure. These paths are those created by other structures that reference the specified structure by use of Execute Structure elements. The Inquire Descendants Of Structure **(GPQPDS)** subroutine returns the paths that lead from the specified structure. The paths are those created by the Execute Structure elements in the specified structure.

In addition to the inquiry subroutines that return structure content, the Inquire Structure Status **(GPQSTS)** subroutine returns to your application an indicator as to whether or not the specified structure exists, and the number of elements in the structure if it does exist.

Element Search

The Element Search **(GPELS)** subroutine is a generalized search facility that operates according to parameters supplied by your application. These included the identifier of the structure to be searched, the search direction (forward or backward), and a list of element types to satisfy the search. By using the element search facility, your application can efficiently locate structure elements for editing. For example, rather than performing a loop that sets the element point and continues to inquire each succeeding element type in search of a particular one, the element search subroutine could perform the search.

Structure Transfer

Two transfer functions allow your application to copy structures from a specified structure store to the currently selected structure store. The source and the target structure stores may reside on the same nucleus or on different nuclei.

To copy one or more structures from a specified structure store, use the Transfer Structures **(GPTST)** subroutine. If you wish to copy all structures from one structure store to the currently selected structure store, use the Transfer All Structures **(GPTAST)** subroutine.

Changing Structure Identifiers and References

At times, applications may find it convenient to give a structure a new identifier. Also, it may be convenient to change any execute structure-type elements (Execute Structure **(GPEXST)** or Conditional Execute Structure **(GPCEXS)**) that reference a structure so that they reference another structure instead. The following subroutines perform these tasks:

- Change Structure Identifier **(GPCSI)**
- Change Structure References **(GPCSRS)**
- Change Structure Identifier and References **(GPCSIR)**

GPCSI changes the specified structure identifier to a new identifier but does not change any references to either the original or the new structure identifiers. The results of using this subroutine are:

- If the two identifiers are the same:
  1. If the structure exists, the call is ignored
  2. If the structure does not exist, a new, empty structure is created.
- If the new structure identifier already exists, then that structure is deleted.
- If the original structure does not exist, then an empty structure is created with the new structure identifier. If the original structure does exist, then its structure identifier is changed to the new structure identifier value.
- If the original structure was the open structure, was associated with a view or workstation, or was referenced by an execute structure-type element then a new, empty structure is created with the original structure identifier value (any execute structure-type elements now reference an empty structure).

Note that any execute structure-type elements that reference the original structure will remain unchanged. If changing a structure identifier results in a structure that references itself (it contains an execute structure-type element that specifies its own structure identifier), then an error occurs and the change is not performed.

GPCSRS changes reference to the original structure identifier so that they reference the new structure identifier, but it does not change any structure identifier. This has the same effect as if your application deleted all execute structure-type elements that refer to the original structure identifier and inserted new execute structure-type elements that refer to the new structure identifier. The results of using this subroutine are:

- If the original structure identifier and the new structure identifier are the same, the call is ignored.
- If the original structure identifier does not exist, the call is ignored.
- If the new structure identifier does not exist then an empty structure is created with the new structure identifier.
- If any structure contains an execute structure-type element that references the original structure identifier, the elements are changed to reference the new structure identifier.
- If changing a structure reference results in a structure that now references itself (it contains an execute structure-type element that specifies its own structure identifier) then an error is issued and the change is not performed.

GPCSIR changes both the original structure identifier to a new structure identifier value and references to the original structure identifier to now reference the new structure identifier value. The result of using this subroutine is as if **GPCSRS** was called followed by a call to **GPCSI** results in a structure that now references itself (it contains an execute structure-type element that specifies its own structure identifier) then an error is issued and the change is not performed.

Conditional Editing

The Conditional Editing **(GPCEDT)** subroutine lets your application improve the efficiency of structure editing by defining a group of editing operations to perform until an error occurs. This eliminates the need to check for operational errors (such as a label not found error when using the Set Element Pointer at Label **(GPEPLB)** subroutine the graPHIGS API will check for the errors and only complete the structure operations if no error occurs.

When Conditional Editing is active, structure editing operations continue to be processed until:

- Conditional editing is ended
- The currently open structure is closed
- An error occurs during an editing operation

Syntax errors do not affect conditional editing and only the operational errors from the following functions terminate conditional editing:

- Copying elements
- Deleting elements
- Moving elements
- Moving the element pointer

When such an error is found, all structure editing functions that require an open structure are ignored until a close structure operation or end conditional editing is encountered.

Structure Store Overflow Prevention

An error which may result in a fatal situation is nucleus storage overflow caused by structure editing. For example, if the storage overflow occurs while defining a sequence of output primitives, recovery from the overflow situation is very difficult because it is difficult for your application to find out how many output primitives have been lost. The Synchronize **(GPSYNC)** subroutine is not suitable for preventing an overflow situation because inserting the **GPSYNC** subroutine between each call to an output primitive routine results in very poor performance due to the many small I/O operations. To solve this problem, a structure store threshold concept is supported by the graPHIGS API Your application can set a threshold value for each structure store and when the structure contents exceeds this value, an event is reported to your application. By setting the threshold to a value smaller than the available storage, your application will be informed of a possible storage overflow situation.

To manage this threshold the graPHIGS API supports the following two subroutines:

- Inquire Available Nucleus Storage **(GPQNCS)**
- Set Structure Store Threshold **(GPSSTH)**

The former returns an estimation of the current free storage on a nucleus and the latter sets the threshold value for each structure store. Note that the storage amount returned by **GPQNCS** is not necessarily exact but will be an estimation within an implementation dependent accuracy. According to the memory management scheme of the nucleus, the value may be greater or less than the exact size of free storage. You must also be aware that the nucleus storage value returned by **GPQNCS** represents the amount of storage available to all application processes connected to the nucleus and therefore does not mean that the entire amount can be used by the application. The amount of storage available to an application

strongly depends on storage usage of other applications. Note that when the value of zero (0) is returned by the **GPQNCS** subroutine, the nucleus storage is unlimited (i.e.a virtual storage system), and no estimation is meaningful.

Also note that only one structure store threshold event is generated, even if your application continues to add or delete data to the structure store. If you want additional structure store threshold events, your application must reset the threshold, using the **GPSSTH** subroutine.

Chapter 13. Archiving Structures

Structure archiving allows your application to store structure information in a file for retrieval at a later time. Upon retrieval, this information is read directly into the structure store in the graPHIGS API nucleus, bypassing the shell. This can result in increased performance and less storage capacity used.

Structure store information is archived in a unique binary format (3) called an *archive file*. An archive file is a graPHIGS API nucleus resource. It must be accessible by the nucleus that is specified on the Open Archive File **(GPOPAR)** subroutine. If the nucleus cannot find the desired archive file (defined by the file descriptor parameter in the **GPOPAR** subroutine), then a new archive file is created by the nucleus. Archive file accessibility is only a potential problem when the nucleus has a `GAM` or `SOCKETS` connection type. No workstation specific information is included in archive files.

Note: On VM, archive files must have a fixed record format with a logical length of 256. On MVS, archive file datasets must have a fixed-blocked record format, a logical record length of 256, and a blocksize of 4096. Any archive files not formatted in this way will cause the message `1205 FILE IS NOT A VALID ARCHIVE FILE` to be sent to the application.

Archive Functions

The archive functions allow your application to store archive files, retrieve, and delete them, as well as make inquiries regarding resource identifiers and potential conflicts. These functions can operate on all current structures or archive files, those associated with a list of root structures, or those listed by identifier.

Opening Archive Files

Your application can open an archive file by using the Open Archive File **(GPOPAR)** subroutine. **GPOPAR** sets the current archive state to Archive Open (AROP), making all archiving functions available. The Attach Resource **(GPATR)** subroutine, used with Resource Type `5=ARCHIVE FILE`, allows one application to access an archive file opened by another application running in a different graPHIGS API shell.

Archiving Files

Archiving transfers an entire structure from the currently selected structure store to a specified archive file. To archive one or more structures, use the Archive Structure **(GPARST)** subroutine to specify individual structures in a list of structure identifiers. The Archive Structure Networks **(GPARSN)** subroutine allows you to specify structures for archiving by root identifier, and the Archive All Structures **(GPARAS)** subroutine archives all structures in the current structure store.

Closing an Archive File

Use the Close Archive File **(GPCLAR)** subroutine to close an open archive file. If no other archive files are open, then the current state becomes archive closed (ARCL) and archiving functions are no longer available.

Retrieving Archive Files

To move one or more structures from a specified archive file to the current structure store, use the Retrieve Structures **(GPRVST)** subroutine. Use the Retrieve Structure Networks **(GPRVSN)** subroutine for all archive files associated with a specified root structure. To move all structures into the current structure store, use the Retrieve All Structures **(GPRVAS)** subroutine.

Deleting Archive Files

Use the Delete Structure from Archive **(GPDSAR)** subroutine to remove one or more structures from an open archive file. Use the Delete Structure Networks from Archive **(GPDSNA)** when you want to remove

all archived structures associated with a root structure. To remove all archived structures from an open archive file, use the Delete All Structures from Archive **(GPDASA)** subroutine.

Inquiring for Archive Files

To determine whether any attached resources contain archive files, use the Inquire List of Attached Resources **(GPQATR)** subroutine, specifying a resource type of `5=ARCHIVE_FILE`.

If an archive file exists in a nucleus, use the Inquire Nucleus Resource Identifier **(GPQNCR)** subroutine to find the resource identifier assigned by the nucleus.

To determine the archive state of the graPHIGS API, use the Inquire Archive State Value **(GPQASV)** subroutine. **GPQASV** returns a value indicating whether or not any archive files are open based on the current archive state (AROP or ARCL).

The Inquire Archive Files **(GPQARF)** subroutine returns a list of open archive file identifiers and associated descriptor information.

Use the Retrieve Structure Identifiers **(GPRSTI)** subroutine to obtain a list of structure identifiers in a specified archive file. Use the Retrieve Identifiers of Structures in Network **(GPRISN)** subroutine to obtain a list of structure identifiers in a specified structure network in an open archive file.

Warning: Temporary Level 4 Header

Ancestors and Descendants of Structures in Archive File: Your application can use two inquiries to determine the structure relationships that precede and follow a particular structure in an archive file. These relationships are expressed as structure identifiers paired with element numbers. For example, the path:

(101,10),(102,5),(103,0)

is interpreted as follows:

- In structure 101, element 10 is an EXECUTE STRUCTURE of structure 102
- In structure 102, element 5 is an EXECUTE STRUCTURE of structure 103
- Structure 103 is the last structure of the path.

The returned data defines the path from structure 101 to structure 103. You can control the order of the path returned, so the top of the path is returned first or the bottom of the path is returned first. In addition, you can control the depth of the path returned (that is, the number of pairs of data returned).

Use the Retrieve Ancestors to Structures **(GPRAS)** subroutine to retrieve the ancestral paths that lead to the specified structure in an archive file. These paths are those created by other structures that reference the specified structure by use of Execute Structure elements. Use the Retrieve Descendants to Structure **(GPRDS)** subroutine to return paths that lead from the specified structure. These paths are those created by the Execute Structure elements in the specified structure in an archive file.

Conflicts

A conflict exists when both the specified archive file and the currently selected structure store contain the same structure identifier. The way in which your application resolves this conflict is determined by the conflict resolution flags in the archive and retrieve subroutines. These flags allow your application to control conflicts by providing three different courses of action:

1=ABANDON
: If a conflict is encountered, then the graPHIGS API exits the function without retrieving or archiving any structure data.

2=MAINTAIN
If a conflict is encountered, then the graPHIGS API does not archive or retrieve the conflicting structure but continues archiving or retrieving all specified structures not in conflict.

3=UPDATE
Archive or retrieve the specified structures regardless of whether a conflict is encountered.

The default for all archiving subroutines is `3=UPDATE` subroutines is `1=ABANDON` To change the conflict resolution flag from the default value, use the Set Conflict Resolution **(GPCNRS)** subroutine. To determine the current value of the flag, use the Inquire Conflict Resolution **(GPQCNR)** subroutine.

Conflict Inquiries

Your application can inquire information regarding potential conflicts by structure or root identifier. The Inquire All Conflicting Structures in Archive **(GPQACA)** subroutine lists conflicting identifiers in the current structure. Inquire Conflicting Structures in Network in Archive **(GPQCNA)** lists conflicting identifiers in the current structure network.

Table 6 shows how the setting of the conflict resolution flag affects archive and retrieve functions under various conditions.

Table 6. Archive and Retrieve Functions and Conflict Resolution

Conditions	Function	Conflict Resolution Flag Values		
		Abandon	**Maintain**	**Update**
Identifier in Source; Identifier not in Destination	Archive	Structure Changed	Structure Changed	Structure Changed
	Retrieve	Structure Changed	Structure Changed	Structure Changed
Identifier in Source; Identifier in Destination	Archive	Error 128	No Change	Structure Changed
	Retrieve	Error 128	No Change	Structure Changed
Identifier not in Source; Identifier not in Destination	Archive	Warning 120 No Change	Warning 120 No Change	Warning 120 No Change
	Retrieve	Warning 120 Empty Structure Created	Warning 120 Empty Structure Created	Warning 120 Empty Structure Created
Identifier not in Source; Identifier in Destination	Archive	Warning 120 No Change	Warning 120 No Change	Warning 120 No Change
	Retrieve	Warning 120 Error 128	Warning 120 No Change	Warning 120 Empty Structure Created

Error 128
Structure conflict occurs when resolution flag is abandon.

Warning 120
One or more structures do not exist.

The Archive File Format

The graPHIGS API uses it own binary file format for archive files (ISO PHIGS does not have a standard binary format for archive files). On VM, archive files must have a fixed record format with a logical length of 256. On MVS, archive file datasets must have a fixed-blocked record format, a logical record length of 256, and a blocksize of 4096.

For VM and AIX, if an archive file does not exist, then one is created by the Open Archive File subroutine **(GPOPAR)** For MVS, however, the dataset for the archive file must be allocated before **GPOPAR** is called.

Estimating Storage Requirements for Archive Files

For MVS, the following can be used as a guide to calculate how much space will be required for an archive file:

- The file header is one record in size.
- For every ten (or fewer) structures in an archive file, one record is allocated for internal control. For example, if there are eleven structures in an archive file, there will be two control records (one to handle the first ten structures and one for the last one, with space for nine more structures).
- An empty structure takes no space in the file other than as an entry in a control record.
- A non-empty structure starts on a record boundary and its space requirements can be calculated in the following manner:
  1. Add up the size of each element in the structure.
  2. Add 12 extra bytes for each grouping of elements that adds up to 65524 or less. Elements are grouped in their order of appearance in the structure and will be grouped together until the next element would cause the element group to be greater than 65524 bytes in size.
  3. Add 8 bytes for every execute structure element or conditional execute structure element. Add another 8 bytes if at least one execute structure-type element exists.
  4. Add 12 bytes for structure information.
  5. Divide the total number of bytes for the structure by 256. The result is the number of records that the structure data uses. If the remainder of the division is not 0, add 1 to the number of records.

The figure, Calculating Space for an Archive File, shows an example calculation of space for an archive file. It assumes that the archive file will hold the following:

- Structure 1 with 16 elements with the following sizes (in bytes):

  8, 8, 8, 10000, 16, 16, 16, 8, 8, 25000, 8, 8, 8, 40000, 16, 16

  Note: The 16-byte elements are execute structure elements.
- Structure 2 with 15 elements with the following sizes:

  12, 8, 8, 8, 15000, 12, 8, 8, 8, 2000, 12, 8, 8, 8, 45000
- Empty Structures 10, 11, 12, ..., 19, 20

Calculating Space for an Archive File

Data Item	Number of Records
File header	1
13 structures exist: 2 control records needed (1 for first 10, 1 for next 3)	2
Structure 1: (Bytes)	
Structure information: 12	
Element group 1:	
Group information: 12	
Element data (13 elements): 35112	
Total: 35124	
Element group 2:	
Group information 12	
Element data (3 elements): 40032	
Total: 40044	
Execute Structure Data:	

```
      Ex.Str. information  8              |
      Ex.Str. data                        |
       (3 elements):      24              |
      ----------------------              |
      Total:                        32    |
   ---------------------------------      |
   Total for structure:          75212     |
                                           |
   75212 / 256 = 293, Remainder = 204      |         294
-------------------------------------------|-------------------
Structure 2:                    (Bytes)    |
                                           |
   Structure information:            12    |
   Element group 1:                        |
      Group information:  12               |
      Element data                         |
       (13 elements):  62108               |
      ----------------------               |
      Total:                        62120  |
   ---------------------------------       |
   Total for structure:          62132     |
                                           |
   62132 / 256 = 242, Remainder = 180      |         243
-------------------------------------------|-------------------
Total records needed in file:              |         540
                                           |

File requires 540 / 16 = 34 blocks of storage
```

Chapter 14. Explicit Traversal Control

Overview

The default processing of the graPHIGS API is to collect primitives and attributes into groups called structures. This grouping allows data to be defined hierarchically and permits easy modification through structure editing operations. In addition, to draw the picture that is defined by the structure content, the graPHIGS API traverses the structures. This traversal makes the displayed picture correct, but requires that all the affected contents be redrawn.

There may be reasons that the structures are *not* a benefit to some applications; such as, the data is not hierarchical or the structure changes are extensive enough as to require complete replacement of all structure content. In addition, you may want to perform your own traversal in your application rather than invoking the graPHIGS API to perform the traversal. (By doing your own traversal, you can realize your own optimizations and ″short-cuts″ that the graPHIGS API does not provide, thus improving end-user responsiveness).

To allow an application to directly control the traversal and drawing process, the graPHIGS API provides subroutine calls that access the very basic rendering controls of the workstation. These controls *suspend* the normal (implicit) processing of the graPHIGS API rendering and allow your application to provide its own (explicit) processing. These controls provide your application explicit traversal control in the following ways:

- ***Immediate Elements***

  Your application can display output primitives on a workstation without requiring the use of structures. Your application sends primitives and attributes directly to a workstation rather than storing them in a structure. Note that *immediate* refers to this bypass of structures: it does not mean that your workstation displays elements immediately in time (although your application can force that result, if desired). An application's use of immediate elements for traversal is often called *Immediate Mode* Your application can use immediate mode in place of the graPHIGS traversal processing of data in structures, or your application can combine immediate mode traversal with traversal of elements in structures (see below).
- ***Direct Traversal Processing***

  Your application can create pictures by directly rendering graphical data in structures. This is similar to immediate mode traversal (in that the application does the traversal operation rather than using the normal graPHIGS traversal processing) but differs in that the application does the traversal of elements stored in structures rather than providing the elements directly. (The application can create a picture by using both immediate mode processing and explicit traversal processing together).
- ***Workstation Resource Control***

  Your application can control the resources used by the workstation during the interpretation of elements for display. These workstation resources are typically frame buffers and Z-buffers used by HLHSR processing. Your application can direct the rendering into the currently displayed frame buffer or to a background frame buffer, it can cause a background frame buffer to be displayed, and it can clear a frame buffer or Z-buffer. These controls allow your application to create pictures, display them, and update the displayed result directly, according to the application's requirements.

Together, these three facilities, Immediate Elements, Direct Traversal Processing, and Workstation Resource Control, are called Explicit Traversal Control, since they suspend the default graPHIGS traversal and allow the application to perform the traversal directly (i.e. explicitly). By using these facilities, a graPHIGS application can implement update techniques that are optimized for the application's requirements, and can use data organizations other than the hierarchical, centralized structure store. For example, an application could:

- add data directly to the display without a complete retraversal
- retraverse only part of a structure to display elements defined by a new attribute element setting (e.g. attribute color)

- render successive steps of an animation into a non-displayed workstation resource, then display the results.

Explicit Traversal Capabilities

Existing graPHIGS applications operate without change and still obtain all the internal optimizations implicitly performed. When using the explicit traversal functions, the application defines its own traversal and can provide its own optimizations in place of those of the graPHIGS API If the workstation is in `WAIT` deferral state and `NO_IMMEDIATE_VISUAL_EFFECT` (NIVE) modification mode, changes are processed only at the application's request. Other combinations of deferral states and modification modes are ignored and the explicit processing forces the workstation into a ″conceptual″ mode of `WAIT/NO_IMMEDIATE_VISUAL_EFFECT`.

Explicit Traversal

All explicit traversals are initiated by the application by calling the Begin Traversal **(GPBGTR)** subroutine, which specifies the workstation to receive the elements produced by the application's traversal. The function parameters include a workstation identifier, a processing type, and processing-type-specific information. This function sets a new workstation state Workstation Selected (WSSL) and can only be called from state Workstation Open (WSOP), which implies **GPBGTR** can not be nested. The selected workstation receives subsequent explicit traversal commands. The workstation becomes dedicated to the traversal processing, and is conceptually put in `WAIT` deferral state and `NO_IMMEDIATE_VISUAL_EFFECT` modification mode, since the application explicitly directs the traversal and rendering processing. (This processing state is similar to the Update Workstation **(GPUPWS)** and Redraw All Structures **(GPRAST)** subroutines, where a traversal is performed on the contents of the structure store. However, **GPBGTR** identifies the workstation to interpret the elements: the application is performing the traversal).

Upon receiving the Begin Traversal **(GPBGTR)** subroutine, the workstation:

- moves the requested WSL fields to the current fields and ensures any changes made to WSL become effective
- initializes the traversal defaults for subsequent traversals.

The application process that issues **GPBGTR** cannot change the contents of the WSL until after an End Traversal **(GPENTR)** subroutine is called. This restriction is to ensure consistency of the rendering of primitives during explicit traversal processing.

GPENTR completes an explicit traversal sequence and requires the structure state Structure Open (STOP) or Structure Close (STCL) and the workstation state Workstation Selected (WSSL). The structure state can not be the newly defined state Non-Retained Structure Open (NROP). Upon receiving this command, the workstation completes any processing of the scene. Notice that this function does not delete the scene or clear the Rendering Resources. The scene can be appended to if the application invokes another **GPBGTR** - **GPENTR** sequence. The effect of appending to an existing scene is Renderer dependent. In addition, the results of appending to a scene are indeterminate if the WSL is changed between **GPENTR** and the next **GPBGTR**.

Immediate Elements

In order for an application to send structure elements directly to a workstation or explicitly traverse structures in the Centralized Structure Store, an application must first call the Begin Structure **(GPBGST)** subroutine. **GPBGST** changes the structure state to Non-Retained Structure Open (NROP) and directs all subsequent elements to the workstation. The End Structure **(GPENST)** subroutine completes the explicit traversal sequence started by **GPBGST** and returns the structure state to Structure Closed (STCL).

GPBGST precedes immediate structure elements that are to be interpreted by a workstation. It is part of the application's traversal processing, and is analogous to the graPHIGS API processing performed by a workstation when traversing a root structure, where the Traversal State List (TSL) is initialized. (Refer to

The graPHIGS Programming Interface: Technical Reference for details on the Travers al State List.) The Begin Structure **(GPBGST)** subroutine does not reference any retained structures nor does it cause elements to be retained. Rather, it indicates that the application is sending elements for interpretation and display by the workstation.

Upon receiving a Begin Structure **(GPBGST)** subroutine call, the workstation:

- moves the requested WSL fields to the current fields and ensures any outstanding changes to the WSL become effective
- initializes the TSL to the traversal defaults assigned by a previous call to the Begin Traversal **(GPBGTR)** subroutine.

A **GPBGST** - **GPENST** sequence is only permitted in workstation state Workstation Selected (WSSL) which implies that they must be bracketed by Begin Traversal **(GPBGTR)** and End Traversal **(GPENTR)** subroutine calls.

Following the **GPBGTR** and **GPBGST** subroutine calls, output primitives and attributes (e.g., Polyline 3 [**GPPL3**]) are passed to the traversing workstation for interpretation. Your application can use any API call that, in the default processing, would create a structure element. (These are the subroutine calls in the Output Primitives and Attributes chapters of *The graPHIGS Programming Interface: Subroutine Reference* manual.) In immediate mode, the same element is sent to the workstation for interpretation rather than being stored in a structure. This includes the Execute Structure element, which results in the traversal of the identified structure. Thus, an application can use immediate mode processing to also traverse structures and their contents. The End Traversal **(GPENTR)** and End Structure **(GPENST)** subroutines indicate completion of the corresponding **GPBGTR** and **GPBGST** subroutines.

The following immediate elements are valid only in state NROP (these are not valid elements to insert into a structure):

- ***Push Set TSL*** **(GPPSTS)** subroutine

  This structure element has a similar effect on the TSL as the Execute Structure **(GPEXST)** subroutine. Unlike the Execute Structure (GPEXST) subroutine, **GPPSTS** does not traverse the specified structure. **GPPSTS** allows an application to perform its own hierarchical traversal. Specifically, this element performs the following:
  - Increments the element counter
  - Pushes the current TSL onto the TSL stack
  - Sets the global modelling transformation to the product of the current global and local modelling transformation matrices
  - Sets the local modelling transformation matrix to the identity matrix
  - Sets the structure identifier entry in the TSL to the one specified by the Push Set TSL structure element
  - Sets the element counter to 0

  If a **GPPSTS** subroutine is encountered when the TSL stack is full, it is treated as a NULL element. However, the workstation maintains a logical stack depth and ignores all structure elements except a Pop TSL **(GPPTS)** subroutine until the stack depth becomes less than or equal to the maximum hierarchy depth. This has the effect of skipping the part of the hierarchy below the maximum hierarchy depth. A warning message is generated each time the logical stack depth is incremented beyond the maximum depth.
- ***Pop TSL*** **(GPPTS)** subroutine

  This structure element removes the top of the stack TSL and copies it to the current TSL. The effect is the same as returning from an Execute Structure **(GPEXST)** subroutine. If the TSL stack is empty when a Pop TSL **(GPPTS)** subroutine is encountered, **GPPTS** is treated as a NULL element and a warning is issued.

Direct Traversal Processing

In the default graPHIGS API processing, traversal is the process of stepping through a structure network, and is performed on the structures associated to a view. The result of traversal is the set of elements (primitives and attributes) to be interpreted by the workstation.

Your application can replace the graPHIGS API default traversal processing with its own traversal. By doing this, your application determines the primitives and attributes to be processed by the workstation and directs the rendering of these elements by the workstation. This processing is called *direct traversal*.

Your application has two methods to provide the elements to the workstation during direct traversal:

- elements can be sent directly to the workstation using the immediate traversal processing, described above
- elements retained in the currently-associated structure store can be explicitly traversed.

The application can use these methods either independently or together to perform the traversal. Thus, direct traversal can be performed using both elements in structures and immediate elements.

Two functions are defined to allow an application to traverse a subset of an entire structure. These two function are the Traversal Elements **(GPTE)** and Accumulate Traversal State **(GPATS)** subroutines. Both functions are only valid in structure state Non-Retained Structure Open (NROP).

The Traversal Elements **(GPTE)** subroutine function causes the traversal of a group of structure elements found in a structure in the associated structure store. Start and end criteria can be specified, allowing the application to render a subset of a structure (for example, traversing the elements between two labels in a structure).

GPATS causes traversal of a group of structure elements found in a structure in the associated structure store. The elements are interpreted only to update the Traversal State List (TSL). No drawing of any primitive elements is performed. Execute Structure elements are ignored, but do increment the element counter. Accumulating the traversal state can be used, for example, to set the TSL before rendering immediate mode primitives. The effect of Accumulate Traversal State is to modify the current TSL which is sometimes called ″Pseudo Traversal″.

Both functions accept a flexible specification for the location in the structure to start and end traversal. These locations may be specified as pick identifier, label, element number, element type, or end-of-structure. An error is generated if the start location does not exist. If the end condition does not exist, traversal stops when the end of the structure is reached.

The element counter in the current TSL is incremented for each element that is traversed by these two functions.

In addition, to the direct traversal subroutines, two other subroutine calls are available for direct processing.

- ***Draw View* (GPDRVW)** subroutine

  This subroutine draws all the roots associated with a view. If the border and the shield of the view are active, then these are also drawn. The specified view is updated as follows:

  – the view table entry is updated by moving the requested entries to the current entries
  – if HLHSR mode is not `OFF`, then the region of any rendering resource (such as a Z-buffer) that corresponds to the view extents is cleared
  – if the shielding indicator is `ON`, then the region of any rendering target (such as a frame buffer) that corresponds to the view extents is cleared to the shielding color
  – elements in the structures associated with the view are traversed.
- ***Draw Image 2* (GPDRI2)** subroutine

This subroutine draws the specified image in a view. The image must be previously defined. Your application specifies the extent of the image and the location, in World Coordinates (WC), where the image is to be drawn.

Control of Workstation Resources

The workstation is responsible for interpreting elements for display on the display surface. The process of converting the elements to displayable data is called **rendering** Each workstation has certain resources that are used during rendering, such as frame buffers, Z-buffers, and memory. The workstation's resources are grouped according to purpose:

- Rendering Targets are the repository for the output of a traversal. A rendering target receives the output generated by rendering primitives with their attributes. Common rendering targets include a frame buffer on a raster display and a file on a CGM or GDF workstation.
- Rendering Resources are resources used by the workstation to assist in the rendering process. The most common rendering resource is a Z-buffer used during HLHSR processing.

A workstation has only one type of rendering target (for example, a frame buffer), although the workstation can have several of them (for example, a double buffered device.) The application can select which of the resources are used during the traversal process. In addition, the application can perform operations such as clearing the resource.

Control of Rendering Targets

Explicit control of the rendering target allows the application to control which target is updated by rendering and when pictures are displayed. There are several functions in this category. All require workstation state Workstation Open (WSOP). These functions include:

- ***Select Rendering Target*** **(GPSRT)** subroutine

  This function selects a Rendering Target on the specified workstation to receive the results of rendering primitives and their bound attributes. For a double-buffered device, this allows rendering to be performed without affecting the current display, or allows rendering to be performed directly to the display surface. This function is only applicable to workstations with two or more Rendering Targets (Implicit updates also change the currently Selected Rendering Target).

  Note: When a workstation is opened, the Selected Rendering Target is the same as the Displayed Rendering Target.
- ***Display Rendering Target*** **(GPDRT)** subroutine

  This function selects a Rendering Target on the specified workstation to be displayed. This function is only applicable to workstations with two or more Rendering Targets and is ignored on workstations with non-displayable Rendering Targets such as a CGM or GDF workstations.
- ***Clear Rendering Target*** **(GPCRT)** subroutine

  This function resets the entire contents of the specified Rendering Targets. For a frame buffer workstation, the pixels are set to the value zero. On devices with a page/frame concept, this function forces a page eject or new frame even if the page/frame is already empty. Clearing a rendering target is typically performed before a traversal to remove any previously rendered output.

Any Rendering Target can be chosen for modification and/or for display. The Rendering Target chosen for modification is called the **Selected Rendering Target** and the Rendering Target chosen for display is called the **Displayed Rendering Target**

For ease in specifying Rendering Targets, they are conceptually organized in a ring as shown in the following figure. To specify a Rendering Target, an application must specify an offset from either the Selected Rendering Target or the Displayed Rendering Target. The offset must be an integer number; positive, negative, or zero. For example, for a double-buffered workstation, an application could select the

displayed buffer plus one for the next update. Upon completion of the update, the displayed buffer is set to the one just drawn (0 relative to the currently Selected Rendering Target).

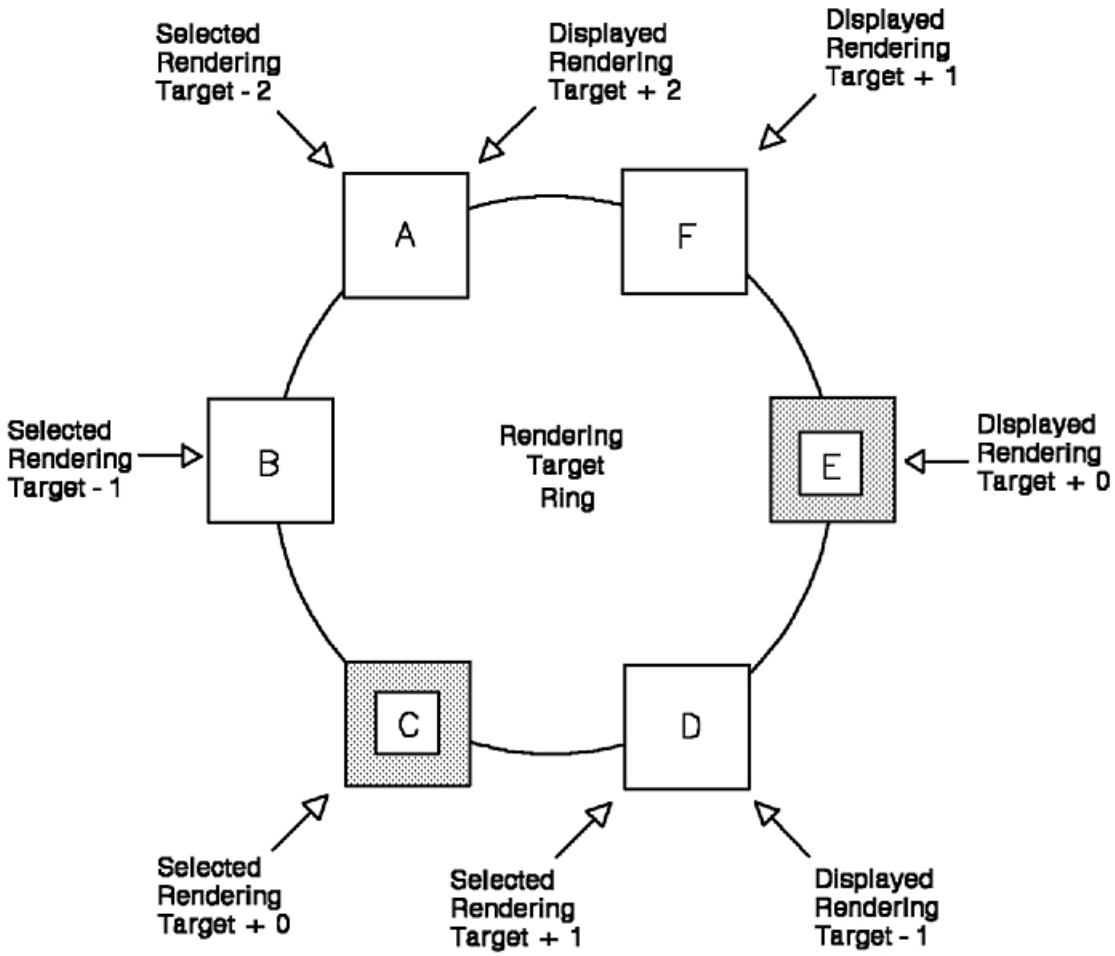


Figure 65. Rendering Target Ring. This illustration shows six squares connected by a circle to form a ring. The squares are labeled A, B, C, D, E, and F starting from the top left side and moving counterclockwise around to the top right side. Square C is the selected rendering target and square E is the displayed rendering target. Square B is labeled as the selected rendering target -1. Square D is labeled as the selected rendering target +1 and the displayed rendering target -1. Square F is the displayed rendering target +1. Square A is labeled as the selected rendering target -2 and the displayed rendering target +2.

Additional resource control subroutines allow your application to control the areas that are defined by the extents of a view:

- ***Copy Viewport* (GPCPVP)** subroutine

  This subroutine copies a region from one rendering target to another rendering target. The region is defined by the viewport of the specified view.

- ***Clear View* (GPCVW)** subroutine

  This subroutine fills a region of a rendering target. The region is defined by the viewport of the specified view. By performing the fill, the previous picture is replaced, thus clearing the area. If the view shielding indicator is `ON`, then the shield color is used to clear the view; otherwise, the view is not cleared. Additionally, if HLHSR mode is `OFF` for the view, then the Z-buffer is not cleared; otherwise, the Z-buffer is cleared.

Control of Rendering Resources

The Clear All Rendering Resources **(GPCARR)** subroutine resets all Rendering Resources for the specified workstation. The function requires workstation state Workstation Open (WSOP) which means that **GPCARR** is not allowed between the Begin Traversal **(GPBGTR)** subroutine and the End Traversal **(GPENTR)** subroutine. This reset occurs in a workstation-dependent manner, depending on the type of resources. For example, if a workstation has a Z-buffer, then **GPCARR** causes the Z-buffer to be cleared.

Inquiry Functions

The Inquire Rendering Targets **(GPQART)** subroutine returns the number of Rendering Targets available on a workstation for explicit use by an application. If the number returned is zero, explicit traversal control is not supported on the workstation.

Effect of Explicit Traversal Functions on Display Surface States

PHIGS defines the following two fields in the WSL to describe the state of the display surface:

- State of Visual Representation,
- Display Surface Empty.

The state of the currently Displayed Rendering Target defines the state of the display surface of the graPHIGS API workstation. In addition, these fields are maintained for every Rendering Target of the workstation.

If a change is made to the WSL or any of the structure networks posted to the workstation, the State of Visual Representation for the currently Selected Rendering Target is set appropriately depending on whether the change was simulated or deferred. The State of Visual Representation is a flag that is set when you post structures to a workstation. Instead of displaying the change to a structure when it is posted, you can make a group of changes and then update the display once. The State of Visual Representation keeps track of whether the display is correct (shows all the changes that have been posted.) (Refer to Frame Buffer Manipulation for an explanation of the State of Visual Representation values.) The State of Visual Representation for the other Rendering Targets is set to `DEFERRED` but their content is not modified.

A Redraw All Structures **(GPRAST)** subroutine results in a `CORRECT` State of Visual Representation for the currently displayed Rendering Target (after completion). That is, the **GPRAST** causes the workstation to be cleared and to redisplay all the posted structures. The Clear Rendering Target **(GPCRT)** subroutine resets the Display Surface Empty status of the specified Rendering Target to `EMPTY` The effect of Update Workstation with control flag set to `PERFORM` is conditional:

- If the State of Visual Representation is `SIMULATED` and the modification mode is `UQUM`, then no update is performed.
- Otherwise, the update is performed and results in a `CORRECT` State of Visual Representation for the currently displayed Rendering Target (after completion).

All explicit traversal control functions set the state of the Selected Rendering Target to `SIMULATED`.

Notice that the display of a different Rendering Target could cause the State of Visual Representation and Display Surface Empty status of the PHIGS workstation to change.

Interactions with Implicit Updates

All of the explicit traversal functions can be used with any combination of deferral states and modification modes. If a change is made to the WSL while in workstation state Workstation Selected (WSSL) and that change requires a modification be made to the display surface, the modification is postponed until the workstation state changes from WSSL to Workstation Open (WSOP). When the workstation state changes to WSOP, then the display is updated if required by the current deferral state and modification mode.

Implicit traversals performed by a workstation are affected by the explicit traversal processing of the application. Implicit regenerations and simulations of updates are not performed while in state WSSL. Implicit correlations and input device echoes are not performed while in state NROP.

When an implicit update traversal is performed, the graPHIGS API changes the contents of the current displayed and selected Rendering Target. The value of the display status is set to `CORRECT` for the displayed Rendering Target, and is unchanged for any other Rendering Targets. After completion of an implicit update, the selected rendering target is the same as the display rendering target.

Considerations When Using Explicit Traversal

There are several important considerations when using the explicit traversal functions. These considerations must be addressed to ensure that your use of explicit traversal does not cause problems with other parts of the graPHIGS API

- ***Explicit Traversal***

  For the most efficient use of the workstation's resources, your application must issue the End Structure **(GPENST)** and End Traversal **(GPENTR)** subroutine calls at reasonable intervals. This allows other processes to use the workstation's resources.

- ***Input Echo***

  Input device echoes cannot be drawn while in state Non-Retained Structure Open (NROP). Your application must issue **GPENST** at reasonable intervals to allow the echo processing to be performed.

- ***Await Event***

  There is a potential for a processing interlock if your application issues an Await Event **(GPAWEV)** subroutine for pick device events while also doing explicit traversal processing. The potential problems occur if the Await Event is issued after a Begin Structure **(GPBGST)** subroutine but before the corresponding End Structure **(GPENST)** subroutine. The interlock occurs because the workstation facilities required to do the (implicit) correlation for the pick device are dedicated to the application's use by **GPBGST** If a pick device is active in `EVENT` mode and is triggered by the user pressing a button, then the workstation queues the pick processing until **GPENST** occurs. However, **GPENST** does not occur while your application is waiting in the Await Event. Note also that input events from other input devices are also queued if a pick event is queued under these circumstances. (This is to preserve the order of input events to the application). Only when the time interval expires does your application receive control again. Since this wait is for the time specified on **GPAWEV**, the workstation's resources are unavailable for other processing.

  It is the responsibility of your application to ensure that such an interlock does not occur. This can be prevented by a variety of methods:

  – do not issue **GPAWEV** while in state NROP when a pick device is in `EVENT` mode. Another way to accomplish the same result is to issue **GPENST** or **GPENTR** as soon as possible. Your application should try to issue these functions as soon as reasonable: this ensures that other parts of the system have access to the resources in order to perform other functions.

  – avoid use of the pick device during explicit traversal processing that dedicates the workstation's resources. Other input devices do not use these resources and they can be used to signal events from the user to the application (as long as the pick device is not used also).

  – use a time value of zero, so that your application never waits if the event queue is empty.

Explicit Traversal Control Examples

This section contains a set of example pseudocode fragments which illustrate the use of Explicit Traversal Control.

Trivial Updates

This example shows the changing of the color of a polyline without forcing a redraw of the entire screen. Through use of the explicit traversal controls, the application can be written to modify the polyline's color

attribute and then retraverse just the attribute and polyline primitive. The code fragment listed below illustrates this type of trivial update, with the following assumptions:

- a structure is posted to the workstation containing a set of Set Polyline Color Index **(GPPLCI)** and Polyline 2 **(GPPL2)** structure element pairs,
- the pseudo variable *structure_id* contains the structure identifier of the structure containing the polylines and color attributes,
- the pseudo variable *element_number* contains the element number within the structure of the polyline that's color is to be changed.

```
.
.
.
CALL GPOPST(structure_id)
  CALL GPEP(element_number-1)
  CALL GPDLE
    CALL GPPLCI(HIGHLIGHT_COLOR)
CALL GPCLST

CALL GPBGTR(WORKSTATION_ID, DRAWING, NULL)
  CALL GPBGST(ANY_INTEGER)
    CALL GPTE(structure_id, START(element_number-1), END(element_number))
  CALL GPENST
CALL GPENTR
.
.
.
```

Accumulate State and Traversal State List Manipulation

The trivial update example can be generalized to use a pick path of any depth by taking advantage of the Accumulate Traversal State **(GPATS)** subroutine. The pseudo code below also illustrates the use of the Push Set TSL **(GPPSTS)** subroutine and the Pop TSL **(GPPTS)** subroutine although, in this particular example, neither are required. The pseudo code was written with the following assumptions:

- a structure exists within a network of structures containing a set of the Set Polyline Color Index **(GPPLCI)** and the Polyline 2 **(GPPL2)** structure element pairs,
- all other polyline attributes are contained in the ancestor structures,
- the pseudo variable `pick_path` contains the pick path to the polyline primitive which is to be highlighted,
- the pseudo variable `depth` contains the depth of the `pick_path`

```
.
.
.
CALL GPOPST(pick_path[depth]
  CALL GPEP(pick_path[depth]
  CALL GPDLE
  CALL GPPLCI(HIGHLIGHT_COLOR)
CALL GPCLST

CALL GPBGTR(WORKSTATION_ID, DRAWING, NULL)
  CALL GPBGST(ANY_INTEGER)
    DO 100 i = 1, depth
      CALL GPPSTS(ANY_INTEGER)
      structure_id   = pick_path[i]
      element_number = pick_path[i]
      CALL GPATS(structure_id, START(ELEMENT1), END(element_number-1))
100 CONTINUE

    CALL GPTE(structure_id, START(element_number), END(element_number))

    DO 200 i = 1, depth
      CALL GPPTS
  200  CONTINUE
  CALL GPENST
```

```
CALL GPENTR
.
.
.
```

Animation and Double Buffering

The example code fragment below illustrates the animation of a line rotating about the origin. The animation is made ″smooth″ through the use of double buffering. The assumptions are as follows:

- the pseudo variable *line* contains the end points of a single polyline segment
- the pseudo variable *angle* contains the current angle of the rotating line measured from the x-axis.

Note: The Inquire Available Rendering Targets **(GPQART)** subroutine function is called to ensure the workstation has at least two Rendering Targets.

```
.
.
.
CALL GPQART(WORKSTATION_TYPE, error_indicator, number_of_rts)

if (number_of_rts.GE> 2) THEN
  line[POINT1]
  line[POINT1]
  loop i = 1 to 360
    CALL GPSRT(WORKSTATION_ID, DISPLAYED_RENDERING_TARGET+1)
    CALL GPCRT(WORKSTATION_ID, SELECTED_RENDERING_TARGET+0)
   CALL GPBGTR(WORKSTATION_ID, DRAWING, NULL)
      CALL GPBGST(ANY_INTEGER)
        angle = angle + DELTA_ANGLE x i
        line[POINT2]
        line[POINT2]
        CALL GPPL2(TWO, line)
       CALL GPENST
    CALL GPENTR
    CALL GPDRT(WORKSTATION_ID, SELECTED_RENDERING_TARGET+0)
.
.
.
```

Redraw All Structures

The Redraw All Structures **(GPRAST)** subroutine function can be defined using the explicit traversal functions as shown in the following pseudo code:

```
CALL GPRAST( workstation_id, flag )
  CALL GPSRT( workstation_id, DISPLAYED_RENDERING_TARGET+1 )

  if ( not_empty(selected_rt) or (flag.EQ.ALWAYS )) THEN
    CALL GPCRT(WORKSTATION_ID, SELECTED_RENDERING_TARGET+0)

  CALL GPBGTR
    for each root
      CALL GPBGST(ANY_INTEGER)
        CALL GPTE(root, ELEMENT_1, END_OF_STRUCTURE)
      CALL GPENST
  CALL GPENTR

  CALL GPDRT(WORKSTATION_ID, SELECTED_RENDERING_TARGET+0)
```

Note: After a call to Redraw All Structures, both the Selected Rendering Target and Displayed Rendering Target are the same.

Update Workstation

The differences between the pseudo code for the Update Workstation **(GPUPWS)** and Redraw All Structures **(GPRAST)** subroutines are as follows:

- the update is performed only if the State of Visual Representation (SVR) of the currently Displayed Rendering Target is DEFERRED or SIMULATED and the control flag parameter is set to PERFORM,
- the Rendering Target is cleared only if it is not already EMPTY

```
CALL GPUPWS( workstation_id, flag )
  IF ( ( state_of_visual_rep(displayed_rt).EQ. DEFERRED or
         state_of_visual_rep(displayed_rt).EQ.SIMULATED ) &
       flag.EQ.PERFORM ) THEN

    CALL GPSRT( workstation_id, DISPLAYED_RENDERING_TARGET+1 )

    IF ( note_empty(selected_rt) ) THEN
      CALL GPCRT(WORKSTATION_ID, SELECTED_RENDERING_TARGET+0)

    for each root
      CALL GPBGST(ANY_INTEGER)
        CALL GPTE(root, ELEMENT_1, END_OF_STRUCTURE)
      CALL GPENST

      CALL GPDRT(WORKSTATION_ID, SELECTED_RENDERING_TARGET+0)
```

Note: If the control flag parameter is set to PERFORM, the State of Visual Representation for the currently displayed Rendering Target upon completion of this function is always CORRECT and the same Rendering Target is both selected and displayed.

Chapter 15. Advanced Viewing Capabilities

The viewing functions provided by graPHIGS API enable you to define or modify views on the World Coordinate System (WCS). Your application must define a view. This involves specifying:

- Orientation through a view matrix
- Clipping volume
- Priority relative to other views
- Characteristics
- Mapping to an output device

These viewing parameters are stored in a view table entry and maintained by the designated workstation. There is one entry per view in this table. If you need to define multiple views, priorities of these views with respect to each other can also be controlled by viewing functions.

The first part of this book (Basic), offered examples and illustrations for many viewing functions. This chapter gives you additional viewing functions that will enable you to increase the application's capabilities and improve performance.

View Priority

Views in the graPHIGS API are related by their input and output priority. View input priority is used to determine which view specification is used to transform LOCATOR and STROKE input from Device Coordinates (DC) to World Coordinates (WC). The decision is made based on the view with the highest relative input priority of all the overlapping views which contain the LOCATOR point or STROKE points. View output priority determines which views are drawn on an output device first. In this case, views with lower priority are drawn first so that higher priority views are drawn on top when overlapping views.

The Set View Priority **(GPVP)** subroutine sets view output and input priorities. This may create problems for some applications. For example, you may wish to display an icon in one corner of a viewport to represent the view orientation within the viewport. The orientation of the icon should change as the view transformation changes, but it should not scale or translate. The icon will always be visible regardless of the content of the view. The easiest way to accomplish this is by using two viewports. The first viewport corresponds to the actual application view. The second overlays the first with higher output priority, and has shielding turned off. Your application would then change the transformation matrix of the second view whenever the first changes, after removing translations and scaling.

Although using two viewports satisfies the output requirements, it may create a problem for input. If you trigger the LOCATOR near the icon (within the icon viewport), the LOCATOR position will be returned with the inverse transform of the icon view, not the actual view of interest. Ultimately, the best solution is to use two unique subroutines for control of input priority and output priority. The first subroutine, Set View Output Priority **(GPVOP)**, controls the relative output priority generated for each view on the display surface. The second, Set View Input Priority **(GPVIP)** subroutine, controls the relative input priority of the viewports for determining which view to use in transforming LOCATOR and STROKE input.

Two inquiry subroutines, Inquire Current View Table Entries **(GPQCVE)** and Inquire Requested View Table Entries **(GPQRVE)**, return the view indexes in the order of decreasing input priority. The Inquire Current View Table Entries Output **(GPQCVO)** and Inquire Requested View Table Entries Output **(GPQRVO)** subroutines return the view indexes in the order of decreasing output priority. The current view table entries will equal the requested entries after a workstation is updated.

View Table Entry

The Set Extended View Representation **(GPXVR)** subroutine enables you to set a view table entry that includes the following items:

- Window clipping indicator
- Near plane clipping indicator
- Far plane clipping indicator
- View shielding indicator
- View border indicator
- View shielding color
- View border color
- Temporary view indicator
- Hidden line/hidden surface removal (HLHSR) mode
- Transparency processing mode
- Initial color processing mode index
- Initial frame buffer protect mask
- Viewport settings
- View volume settings
- View matrix
- View input activity flag
- View output activity flag
- View mapping matrix
- Antialiasing mode
- Shield alpha value

Following is a discussion of each of the items listed above:

Window Clipping Indicator

The window clipping indicator for a view can be either `ON` or `OFF` and specifies whether or not primitives are clipped to the window boundary of a view.

Near and Far Plane Clipping Indicator

In addition to the clipping which can occur at the window boundary, clipping can also be performed at the near and far clipping plane of a view under the control of the Near and Far Clipping Plane Indicator. Near and far clipping planes are discussed in detail in Chapter 5. Viewing Capabilities.

View Shielding Indicator and Color

The background or ″shield″ of a view can be turned `ON` or `OFF` with the View Shielding Indicator. When turned `ON`, a view's background is filled with the View Shielding Color which can be specified as an index into the workstation's rendering color table or as a direct color vector. A view's background shield provides a means for your application to block the contents of overlapping lower priority views. The use of view shielding can also improve the performance of your application when views overlap due to the use of view optimization by the graPHIGS API.

View Border Indicator and Color

The border of a view can be turned `ON` or `OFF` using the View Border Indicator. When `ON`, the border is drawn in the color specified by the View Border Color.

Temporary View Indicator

In some applications, pop-up areas, such as pop-up menus, are used to interact with the user. These areas are typically active for a relatively short period of time and usually have the highest or near highest display priority. Also, these areas usually have the user's attention so there will be few changes to other regions of the display.

The content of these pop-up areas is usually simple, yet they may be comprised of any combination of primitives and attributes. Through the graPHIGS API, pop-up areas are best simulated through the use of views where each pop-up area corresponds to a graPHIGS API view with shielding turned on. The initial display performance of these areas is typically quite good due to the redraw optimization techniques provided by the graPHIGS API However, the deactivation or removal of a pop-up area may take a long time depending on the complexity of the underlying views.

To improve the performance of deactiviating a pop-up area, a view can be marked as ″temporary″ through the Temporary View Indicator. Views marked as temporary can then be processed in a workstation-dependent way to optimize the deactivation of the view.

Hidden Line/Hidden Surface Removal

The use of Hidden Line/Hidden Surface Removal (HLHSR) in each view can be controlled by your application through the Hidden Line/Hidden Surface Removal mode. To determine which HLHSR modes are supported on a particular workstation, use the Inquire Available HLHSR Modes **(GPQHMO)** subroutine. A complete discussion of HLHSR is contained in Chapter 16. Rendering Pipeline.

Transparency Processing

Like HLHSR, the transparency processing of a view may be controlled by your application. To determine which transparency modes are supported on a particular workstation, use the Inquire Available Transparency Modes **(GPQTMO)** subroutine. A complete discussion of transparency processing is contained in Transparency. See also Shield Alpha Value.

Color Processing

You can control the Initial Color Processing Model of a view through the Initial Color Processing Index, which is an index into the workstation's color processing model table. This control determines the mode used for the display of images mapped to the view, as well as the traversal default for structures associated to the view. The support for color processing on a particular workstation can be inquired using the Inquire Color Processing Facilities **(GPQCPF)** subroutine. For more information on color processing, see Chapter 17. Manipulating Color and Frame Buffers.

Note: The Initial Color Processing Model does not affect the view's shielding or border colors. These colors are always quantized using color processing table entry zero.

Frame Buffer Protect Mask

A frame buffer mask may be used to protect individual frame buffer bit planes within the specified view. For more information on frame buffers and frame buffer masks, see Chapter 17. Manipulating Color and Frame Buffers.

Viewport Settings

Your application can set the 2D or 3D viewport of a view which controls where the view will be displayed in Normalized Projection Coordinates (NPC). For more information concerning viewport setting, see Chapter 5. Viewing Capabilities.

View Volume Settings

A viewing volume specifies the region of World Coordinate (WC) space which will be visible through a graPHIGS API view. The volume indicates the following information:

- the limits on the view window
- the position of the view near and far clipping planes
- the viewing plane distance
- the location of the projection reference point relative to the Viewing Coordinate System (VCS)
- the view projection type (parallel or perspective)

For more information concerning viewing volumes, see Chapter 5. Viewing Capabilities.

View Matrix

Your application can use a 2D or 3D view matrix to transform an object's coordinates from the World Coordinate System (WCS) to the Viewing Coordinate System (VCS). A 2D matrix is specified as a 3 [default] 3 homogeneous matrix, and the graPHIGS API expands it to a 4 [default] 4 matrix.

View Activity

A view can be defined as active or inactive for both input or output. When active for input, a view can be used to transform locator and stroke data from Device Coordinates (DC) to World Coordinates (WC). The contents of a view which is active for output will be drawn on the display surface when the workstation is updated.

The view icon problem discussed at the beginning of this chapter in the view priority section can also be solved using view activity. In this case, the view containing the view orientation icon would be activated for output only, and the other view would be activated for both input and output. By activating the icon view for output only, the view will not be considered for transforming locator or stroke input.

View Mapping Matrix

To transform an object's coordinates from Viewing Coordinates to Normalized Projection Coordinates (NPC), a view mapping matrix is used. This transformation can also be specified using the view volume settings, as discussed previously. The view mapping matrix is simply an alternative method.

In the viewing process, transforming from WCS to VCS is processed by a view matrix. The view mapping matrix is then used to transform the coordinates from VCS to NPC coordinates. Your application can create a view mapping matrix using the Evaluate View Mapping Matrix subroutines **(GPEVM3** and **GPEVM2)**.

Antialiasing Mode

The view's Antialiasing Mode controls the effect of a structure's antialiasing identifier attribute. If the view's mode is `OFF`, then no antialiasing is performed, otherwise, antialiasing is performed as specified by the antialiasing attribute. Two modes of antialiasing are available. Antialiasing Mode `2=SUBPIXEL_ON_THE_FLY` results in the highest quality rendering. `3=NON_SUBPIXEL_ON_THE_FLY` provides better performance. Antialiasing techniques are workstation dependent and may not be available on all workstations. Refer to the workstation description information in *The graPHIGS Programming Interface: Subroutine Reference* for a description of available antialiasing methods and restrictions. Use the Inquire Available Antialiasing Modes **(GPQAMO)** to determine the antialiasing capabilities available on your workstation.

Shield Alpha Value

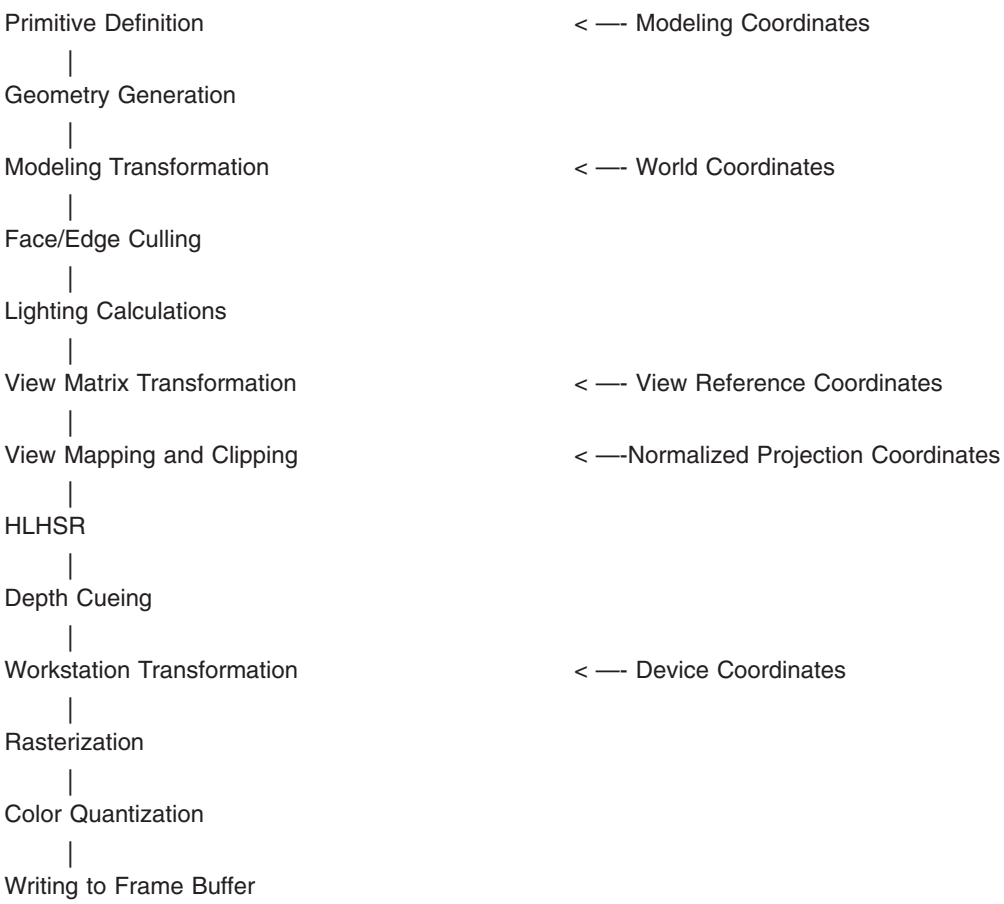
The shield alpha value is used in the transparency blending operation and is expressed as an integer in the range of 0 to 255. You use this value to initialize the destination alpha values ($alpha_{dest}$) when alpha planes are present and the shielding indicator is on. The initial shield alpha value does not effect the initial

shield color, but is used to blend subsequent primitives with the view shield when blending is in effect. If no value is specified, then the value of 255 is used for the initial alpha value. See Transparency for a discussion of the blending function.

Chapter 16. Rendering Pipeline

This chapter provides an overview of the graPHIGS API processing performed on primitives in order to generate output on a display surface. The sequence of processes applied to primitives in a graphics package is called its *rendering pipeline*. The figure below shows each step in the graPHIGS API rendering pipeline process. This chapter explains the basic operation performed by each of these steps.

The graPHIGS API Rendering Pipeline



Following is a brief definition of each of the rendering steps. The advanced rendering concepts are discussed in detail in the remainder of this chapter and the next.

Rendering step:	Definition:	See this section for additional information:
Primitive Definition	Geometric primitives are defined when your application creates primitive structure elements, such as polylines, polygons, NURBS curves and surface. Primitive definition is discussed in detail in the Chapter 4. Structure Elements and Chapter 11. Structure Elements.	
Morphing	Modifies the geometry and/or data (texture) mapping data of a primitive.	Morphing

Rendering step:	Definition:	See this section for additional information:
Geometry Generation	From a given primitive definition, a set of geometric entities are generated. The types of entities that are generated and their characteristics are determined by the geometric attributes of the primitive. For some entities, such as curves and surfaces, this step may be deferred to later in the pipeline after transformations have been performed on control points. This is done only if no distortion to lighting calculation is introduced by the transformation.	Geometry Generation
Modeling Transformation	Transforms the geometric entities by the matrix resulting from the concatenation of the current local and global modeling transformation matrixes.	
Modeling Clipping	Provides a way to clip geometric entities in world coordinates.	Modeling Clipping
Face/Edge Culling	Determines whether a given geometry should be further processed or not based on its geometric information. In most cases, the geometry examined in this step will be the normals of planar surfaces. This step can, for example, reject a primitive based on its normal.	Face and Edge Culling
Lighting Calculations	Simulates the effects of light sources illuminating the primitives. This process is performed only on area geometries.	
Data (Texture) Mapping	Uses data values from the vertices of primitives to determine the colors to be used to render the primitives. *Texture Mapping* is a method of filling the interior of area primitives from a collection of colors.	
View Matrix Transformation	Transforms the resulting geometry to view reference coordinates by using the current view orientation matrix.	
View Mapping and Clipping	Performs the window to viewport mapping and perspective projection if specified. Primitives are also clipped to the view volume.	
Hidden Line/Hidden Surface Removal (HLHSR)	Determines the geometric relationship between geometries within the scene. Based on the relation, it is decided whether a given geometry should be further processed or not.	Hidden Line/Hidden Surface Removal (HLHSR)

Rendering step:	Definition:	See this section for additional information:
Depth Cueing	Changes the color of geometries based on their z-location within Normalized Projection Coordinate (NPC) space. This helps give the generated picture an illusion of depth.	
Workstation Transformation	Transforms the geometry of the primitive to device coordinates after being clipped to the workstation window.	
Rasterization	Determines final shape of a given geometry by using its rendering attributes, such as line style, etc. The geometric shape is digitized to a set of pixel coordinates.	
Transparency	Allows you to control the degree to which you can ″see through″ a primitive.	
Color Quantization	Maps colors generated by the previous stages to those available on the workstation based on application specified controls. For detailed information on color quantization, see Chapter 17. Manipulating Color and Frame Buffers.	
Writing to Frame Buffer	Places the rendered output into the frame buffer, possibly using a frame buffer operation and/or a masking operation. For detailed information on frame buffer operations, see Chapter 17. Manipulating Color and Frame Buffers.	

Morphing

Overview

Morphing lets you transform an object from one shape to another where the start and end shapes have the same number of vertices. The term itself is derived from the words metamorphosis or metamorphose, which mean changed or to change in form. You define the changes using morphing values at each vertex of the affected primitives along with scale factors that define how the vertex morphing values affect the primitive.

There are two types of morphing, one using vertex coordinates and the other using data mapping data values. These functions are called ***vertex morphing*** and ***data morphing***, respectively.

You can use vertex morphing with the following primitives:

- Polyline Set 3 with Data **(GPPLD3)**
- Polygon 2 with Data **(GPPGD2)**
- Polygon 3 with Data **(GPPGD3)**
- Triangle Strip 3 **(GPTS3)**
- Quadrilateral Mesh 3 **(GPQM3)**

You can use data morphing with the following primitives:

- Polygon 2 with Data **(GPPGD2)**
- Polygon 3 with Data **(GPPGD3)**
- Triangle Strip 3 **(GPTS3)**
- Quadrilateral Mesh 3 **(GPQM3)**

Because data morphing applies to area primitives only, data morphing cannot be used for polylines.

Note: Use the Inquire Workstation Description **(GPQWDT)** call to determine whether your workstation supports the graPHIGS API morphing facilities.

Vertex Morphing

Vertex morphing allows you to modify the rendered geometry of the primitive without changing the structure element. To achieve the effects of morphing, your application must provide ***morphing vectors*** with each vertex in the primitive definition. These act together with the ***morphing scale factors*** to modify the rendered primitive. You use the Set Vertex Morphing Factors **(GPVMF)** subroutine to specify the scale factors. Vertex morphing vectors and scale factors affect only the vertex coordinates, while data morphing vectors and scale factors affect only the data mapping data values at each vertex.

Vertex morphing takes place in modeling coordinates. The modeling coordinate values specified in the primitives are modified by the vertex morphing process to produce new modeling coordinate values. These modified coordinate values are then used in the graPHIGS API pipeline processing to render the primitive. The values are used only during traversal and do not update any element content.

- ***Polygons***: Polygon primitives are planar primitives, and the rendering results are undefined if all the points do not lie in the same plane. With vertex morphing you could cause polygons that are defined as planar in Modeling Coordinates to become non-planar (after the addition of the vertex morphing terms). To avoid undesirable rendering results when using vertex morphing with polygon and quadrilateral mesh primitives, your application should ensure that all vertex morphing operations result in planar polygons, or that all affected polygons define only triangles which by definition are always planar.
- ***Normal Vectors***: Normals and other vertex values remain unchanged even if you modify the vertex coordinates through vertex morphing. If you specify normals with a primitive, they will be used as specified regardless of the change in geometry. This could cause unexpected lighting results. However, if you do not specify normals with a primitive and lighting is enabled, graPHIGS API automatically calculates a geometric normal for each primitive facet. (See Geometry Generation. These normals are calculated after morphing, and reflect the modified geometry.

Data Morphing

Data morphing occurs in data space and results in morphed data values. That is, data morphing occurs before all transformations are applied to the data values in the graPHIGS API rendering pipeline. You use the Set Data Morphing Factors **(GPDMF)** subroutine to specify the data morphing scale factors.

Data morphing changes the vertex data mapping values, before the data matrix is applied. See Set Data Matrix 2 **(GPDM2)** and Set Back Data Matrix 2 **(GPBDM2)** procedures in *The graPHIGS Programming Interface: Subroutine Reference* for more information on the data matrix specification. You can only specify Data morphing values if corresponding data mapping data values exist in the primitive definition.

Data morphing can be interpreted in different ways, depending on the data mapping method being used. See Texture/Data Mapping. If you use data mapping for contouring to represent application-specific values such as temperature, then data morphing reflects temperature changes that may change the interior color of the primitive. If data mapping is used for texture mapping, then data morphing provides a means to stretch the texture image across the primitive.

The Morphing Equation

In vertex morphing, the vertex coordinate values (x, y, z) combine with the vertex morphing scale factors (s_1, s_2, ..., s_{nscale}) and the vertex morphing vectors ((dx_1, dy_1, dz_1), (dx_2, dy_2, dz_2), ..., ($dx_{nvector}$, $dy_{nvector}$, $dz_{nvector}$)) to create the new vertex coordinate values (x', y', z') as follows:

$$x' = s_1x + s_2dx_1 + s_3dx_2 +...+ s_{nscale}dx_{nvector}$$

$$y' = s_1y + s_2dy_1 + s_3dy_2 +...+ s_{nscale}dy_{nvector}$$

$$z' = s_1z + s_2dz_1 + s_3dz_2 +...+ s_{nscale}dz_{nvector}$$

In data morphing, the data mapping data values (x_1, x_2, ..., x_{ndata}) combine with data morphing scale factors (s_1, s_2, ..., s_{nscale}) and data morphing vectors (($d_{1,1}$, $_{1,2}$, ..., $d_{1,ndata}$), ($d_{2,1}$, $_{2,2}$, ..., $d_{2,ndata}$), ..., ($d_{nvector,1}$, $d_{nvector,2}$, ..., $d_{nvector,ndata}$)) to create the new data mapping data values (x'_1, x'_2, ..., x'_{ndata}). This combination is of the form:

$$x'_1 = s_1x_1 + s_2d_{1,1} + s_3d_{2,1} +...+ s_{nscale}d_{nvector,1}$$

$$x'_2 = s_1x_2 + s_2d_{1,2} + s_3d_{2,2} +...+ s_{nscale}d_{nvector,2}$$

...

$$x'_{ndata} = s_1x_{ndata} + s_2d_{1,ndata} + s_3d_{2,ndata} +...+ s_{nscale}d_{nvector,ndata}$$

Number of Scale Factors: These equations show that the number of morphing scale factors should be one more than the number of morphing vectors in the affected primitive (*nscale* = *nvector* + 1) if the number of morphing vectors and scale factors disagree at traversal time, then zero-value vectors or scale factors are assumed wherever necessary. That is, if you supply too many scale factors for a given primitive (*nscale* > *nvector* + 1), then the graPHIGS API ignores the extra scale factors as if there were additional zero-valued morphing vectors in the primitive definition. And, if too few scale factors are supplied (*nscale* < *nvector* + 1), the extra morphing vectors are ignored, as if there were additional scale factors supplied with value 0.

Scale Factor Constants: You supply data morphing scale factors as a list of parameters using Set Data Morphing Factors **(GPDMF)** and Set Back Data Morphing Factors **(GPBDMF)** subroutines.

You supply vertex morphing scale factors as a list of parameters using the Set Vertex Morphing Factors **(GPVMF)** subroutines.

You can query the maximum number of morphing vectors (and parameters) a particular workstation supports by using the Inquire Workstation Description **(GPQWDT)** subroutine.

Morphing Examples

You can use morphing to perform several different types of modifications to the rendered primitive values. This section contains a few examples.

Metamorphosis - Linear over Time

Morphing transforms one object's geometry into another, provided the same number of vertices are used to define both objects. The metamorphosis outlined in this section causes each vertex in the initial object to follow a linear path to the corresponding vertex in the final object.

1. Specify both object locations with each vertex. Let ($x_{initial}$, $y_{initial}$, $z_{initial}$) represent the location of a vertex at the initial time ($t_{initial}$), and (x_{final}, y_{final}, z_{final}) represent the same vertex at the final time (t_{final}). Then set the vertex coordinates and a single vertex morphing value:

   $$(x, y, z) = (x_{initial}, y_{initial}, z_{initial})$$

$(dx_1, dy_1, dz_1) = (x_{final}, y_{final}, z_{final})$

2. For each time (t) between the initial and final times ($t_{initial} <= t <= t_{final}$), define u to represent the *normalized* time ($0.0 <= u <= 1.0$):

$$u = \frac{(t - t_{initia})}{(t_{fina} - t_{initia})}$$

Figure 66. .

Then set the vertex morphing factors:

$(s_1, s_2) = ((1 - u), u)$

so that:

$(s_1, s_2) = (1.0, 0.0)$ at time $t_{initial}$

$(s_1, s_2) = (0.0, 1.0)$ at time t_{final}

Substituting these values in the vertex morphing equations, you see that $(x, y, z) = (x_{initial}, y_{initial}, z_{initial})$ at time $t_{initial}$ and $(x, y, z) = (x_{final}, y_{final}, z_{final})$ at time t_{final}, as shown in the following figure:

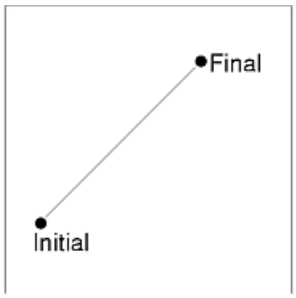


Figure 67. First Order Linear Over Time. This illustration shows two points with a diagonal, straight line connecting them. This diagonal goes from the lower left to the upper right. The point on the lower left is labeled Initial and the point on the upper right is labeled Final.

Metamorphosis - Second Order Curve over Time

The metamorphosis outlined in this section causes each vertex in the initial object to follow the path of a second order Bezier curve to the corresponding vertex in the final object.

1. Specify both object locations and an additional control point with each vertex. Let $(x_{initial}, y_{initial}, z_{initial})$ represent the location of a vertex at the initial time ($t_{initial}$), $(x_{final}, y_{final}, z_{final})$ represent the same vertex at the final time (t_{final}), and $(x_{control}, y_{control}, z_{control})$ represent the vertex control point. Then set the vertex coordinates and two vertex morphing vectors:

   $(x, y, z) = (x_{initial}, y_{initial}, z_{initial})$

   $(dx_1, dy_1, dz_1) = (x_{control}, y_{control}, z_{control})$

   $(dx_2, dy_2, dz_2) = (x_{final}, y_{final}, z_{final})$

2. For each time (t) between the initial and final times ($t_{initial} <= t <= t_{final}$), define u to represent the *normalized* time ($0.0 <= u <= 1.0$):

$$u = \frac{(t - t_{initia})}{(t_{fina} - t_{initia})}$$

Figure 68. .

Then set the vertex morphing factors:

$(s_1 , s_2 , s_3) = ((1 - u)^2 , 2u(1 - u), u^2)$

so that:

$(s_1 , s_2 , s_3) = (1.0, 0.0, 0.0)$ at time $t_{initial}$

$(s_1 , s_2 , s_3) = (0.0, 0.0, 1.0)$ at time t_{final}

Substituting these values in the vertex morphing equations, you see that $(x, y, z) = (x_{initial}, y_{initial}, z_{initial})$ at time $t_{initial}$, and $(x, y, z) = (x_{final}, y_{final}, z_{final})$ at time t_{final}, as shown in the following figure:

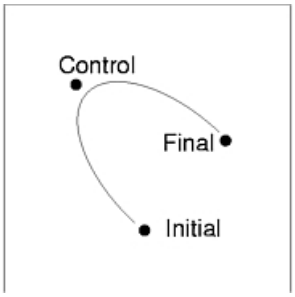


Figure 69. Second Order Bezier Curve Over Time. This illustration shows a curved line connecting two points and the control point for the curve. The first point on the curve is labeled Initial and the second point is labeled Final. The control point, labeled Control, is located outside of the middle of the curve.

Metamorphosis - Third Order Curve over Time

The metamorphosis outlined in this section causes each vertex in the initial object to follow the path of a third order Bezier curve to the corresponding vertex in the final object.

1. Specify both object locations and two additional control points with each vertex. Let $(x_{initial}, y_{initial}, z_{initial})$ represent the location of a vertex at the initial time $(t_{initial})$, $(x_{final}, y_{final}, z_{final})$ represent the same vertex at the final time (t_{final}), and (x_{c1}, y_{c1}, z_{c1}) and (x_{c2}, y_{c2}, z_{c2}) represent the two control points. Then set the vertex coordinates and three vertex morphing vectors:

   $(x, y, z) = (x_{initial}, y_{initial}, z_{initial})$

   $(dx_1, dy_1, dz_1) = (x_{c1}, y_{c1}, z_{c1})$

   $(dx_2, dy_2, dz_2) = (x_{c2}, y_{c2}, z_{c2})$

   $(dx_3, dy_3, dz_3) = (x_{final}, y_{final}, z_{final})$
2. For each time (t) between the initial and final times ($t_{initial} <= t <= t_{final}$), define u to represent the *normalized* time ($0.0 <= u <= 1.0$):

$$u = \frac{(t - t_{initia})}{(t_{fina} - t_{initia})}$$

Figure 70. .

Then set the vertex morphing factors:

$(s_1, s_2, s_3, s_4) = ((1 - u)^3, 3u(1 - u)^2, 3u^2(1 - u), u^3)$

so that:

$(s_1, s_2, s_3, s_4) = (1.0, 0.0, 0.0, 0.0)$ at time $t_{initial}$

$(s_1, s_2, s_3, s_4) = (0.0, 0.0, 0.0, 1.0)$ at time t_{final}

Substituting these values in the vertex morphing equations, you see that $(x, y, z) = (x_{initial}, y_{initial}, z_{initial})$ at time $t_{initial}$, and $(x, y, z) = (x_{final}, y_{final}, z_{final})$ at time t_{final}, as shown in the following figure:

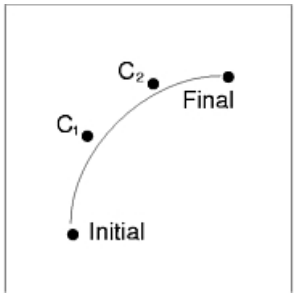


Figure 71. Third Order Bezier Curve Over Time. This illustration shows a curved line connecting two end points with two defined control points, C_1 and C_2. The first end point is labeled Initial and the second end point is labeled Final. The control points are located outside the curve, and each one is evenly spaced between the two end points.

Data Contour Modification

Morphing also provides a means to represent changes in application-specific data, which causes variations in a primitive's data mapping *contour bands* the contouring changes outlined in this section allow a single data value, namely temperature, to vary from one set of readings to another. These temperature changes are linear over time; however, non-linear data changes may also be accomplished in a manner similar to the vertex morphing examples above.

1. Specify both temperature readings with each vertex. Let $temperature_{initial}$ represent the temperature of a vertex at the initial time ($t_{initial}$), and $temperature_{final}$ represent its temperature at the final time (t_{final}). Then set the vertex data mapping value and one data morphing vector:

   $x_1 = temperature_{initial}$

   $d_1 = temperature_{final}$

2. For each time (t) between the initial and final times ($t_{initial} <= t <= t_{final}$), define u to represent the *normalized* time ($0.0 <= u <= 1.0$):

$$u = \frac{(t - t_{initia})}{(t_{fina} - t_{initia})}$$

Figure 72. .

Then set the data morphing factors:

$(s_1, s_2) = ((1 - u), u)$

so that:

$(s_1, s_2) = (1.0, 0.0)$ at time $t_{initial}$

$(s_1, s_2) = (0.0, 1.0)$ at time t_{final}

Substituting these values in the data morphing equations, you see that $x_1 = temperature_{initial}$ at time $t_{initial}$, and $x_1 = temperature_{final}$ at time t_{final}.

Geometry Generation

The graPHIGS API includes many functions to process primitives based on their geometric data. In other words, assume that all geometric data is conceptually treated as one of the following:

1. **Glyph**

   This geometric entity is mathematically a point and so it has no spatial measure. However on the actual display surface, it is displayed as a set of pixels with a fixed relationship. Therefore on most workstations, it is treated as a planar object on a plane parallel to the display surface. This geometry is generated by the following primitives. Note that many of the listed primitives may generate more than one glyph.

   Polymarker 2/3
   Marker Grid 2/3
   Annotation Text 2/3
   Annotation Text Relative 2/3
   Pixel 2/3

2. **Line**

   A straight line connecting two endpoints. Like the glyph, it has no spatial measure but is displayed as a set of pixels approximating the mathematical line. Therefore, on most workstations it is treated as a planar object on a plane which includes the mathematical line. The plane will be a plane parallel to the display when the line itself is parallel to the display. Otherwise, the plane is a plane which intersects with the display surface on a horizontal or vertical line. This geometry is generated from the following primitives:

   Polyline 2/3
   Disjoint Polyline 2/3
   Polyline Set 3 With Data
   Polyhedron Edge
   Line Grid 2/3
   Circle 2
   Circular Arc 2
   Ellipse 2/3
   Elliptical Arc 2/3
   Non-Uniform B-Spline Curve 2/3

   and by the following primitives when they have the edge flag attribute on:

   Polygon 2/3
   Polygon With Data 2/3
   Composite Fill Area 2
   Triangle Strip 3
   Quadrilateral Mesh 3
   Non-Uniform B-Spline Surface
   Trimmed Non-Uniform B-Spline Surface

For the quadrilateral mesh or the polygon with data primitive, the application can individually control whether a given edge generates a line entity or not by the boundary flags in the primitive definition.

3. **Area**

   A semi planar region surrounded by a closed sequence of lines. The term “semi” means that the region may not be exactly planar in some cases but is treated as planar. This geometric entity is generated by the following primitives:

   Polygon 2/3
   Polygon With Data 2/3
   Composite Fill Area 2
   Quadrilateral Mesh 3
   Triangle Strip 3
   Non-Uniform B-Spline Surface
   Trimmed Non-Uniform B-Spline Surface

4. **Text**

   A set of co-planar geometries which represent a sequence of character shapes. On some workstations, these geometries may be processed as if they are line or area geometries. Geometries of this category are generated only by the following primitives:

   Text 2/3
   Character Line 2

Geometric Attributes

Each primitive has many attributes. Some of them are used to generate geometric entities described in the previous section and affect the geometric processing described in this chapter. Others are bound to the entities and will be used for purposes other than geometric processing. We call attributes in the first group geometric attributes.

As described in the previous section, geometric entities generated by a given primitive may be different for each workstation. Therefore, a set of geometric attributes may also be workstation-dependent. For example, on a workstation which treats a line geometry as a planar object with some width, the line width scale factor attribute may affect the geometric processing and so it may be considered to be a geometric attribute.

To minimize such workstation dependencies, we define a set of attributes which are always treated as geometric attributes and another set of attributes which may be treated as geometric attributes or not depending on the workstation. Any workstation supported by the graPHIGS API products will never use other attributes as geometric attributes and so they will not affect the geometric processing described here.

Table 7. Workstation-dependent geometric attributes.

PRIMITIVE GROUP	ATTRIBUTE	WORKSTATION DEPENDENT?
Polymarker	Marker Type Marker Size Scale Factor	Yes Yes
Polyline	Line Width Scale Factor	Yes
Polygon	Edge Flag Edge Line Width Scale Factor	No Yes

PRIMITIVE GROUP	ATTRIBUTE	WORKSTATION DEPENDENT?
Geometric Text	Character Height	No
	Character Up Vector	No
	Text Path	No
	Text Alignment	No
	Text Font	No
	Text Precision	No
	Character Expansion Factor	No
	Character Spacing	No
	Character Positioning Mode	No
Annotation Text	Annotation Height Scale Factor	Yes
	Annotation Height	Yes
	Annotation Up Vector	Yes
	Annotation Path	Yes
	Annotation Alignment	Yes
	Annotation Style	Yes
	Text Font	Yes
	Text Precision	Yes
	Character Expansion Factor	Yes
	Character Spacing	Yes
	Character Positioning Mode	
Note: The name set attribute is not considered as a geometric attribute but it will affect whether a given primitive generates geometric entities or not through the invisibility filter of a workstation. When a given primitive becomes invisible by the name set and filter mechanism, it does not generate any geometric entity and is treated as not existing for all geometric processing.		

Curve Tessellation

On some workstations curves are divided into sets of line geometries before they are processed by the rendering pipeline. Such a process is called **curve tessellation** The application can control the process by the curve approximation criteria attribute in conjunction with its associated approximation value and possibly the tessellation vector specified in a curve definition itself.

The curve approximation criteria are defined in Chapter 11. Structure Elements.

Note: The curve approximation criteria attribute is applied to the non-uniform curve primitives but is not applied to the circle, ellipse and arcs.

Surface Tessellation

As with curves, your application can control the tessellation of surfaces using the Set Surface Approximation Criteria attribute. The Set Surface Approximation Criteria **(GPSAC)** structure element defines two approximation values, one for the u and one for the *v* directions of the surface. The meaning of the approximation values are conceptually the same for curves and surfaces.

From the definition of the surface approximation criteria, a surface is tessellated into a set of area geometries each of which has a rectangular shape in the topological sense. However, a workstation may further divide the rectangle into a set of smaller pieces, typically into two triangles.

In addition to the surface approximation criteria, the tessellation of a trimmed NURBS surface near a set of trimming curves is also controlled by the Set Trimming Curve Approximation Criteria **(GPTCAC)** structure element.

When creating curves, you can control the tessellation process of a surface into a set of area geometries with the surface approximation criteria attribute. The same set of approximation criteria for curves is defined for surfaces. However, the Set Surface Approximation Criteria **(GPSAC)** structure element has two

associated approximation values each of which specifies an approximation value to be applied to one of two parameters of the curve definition. The first and second values are used for the first *u* and second *v* parameters, respectively.

Geometric Normal

Each area geometry has a geometric normal defined in modeling coordinates. It is defined according to each area defining primitive as follows:

2D Primitives

Any area geometry generated by 2D primitives has a geometric normal (0,0,1).

Polygon 3

This primitive generates one or more area geometries and all of them have the same geometric normal. It is calculated from coordinates (MC) of the first three vertices of the primitive definition by using the following equation:

$$\mathbf{N} = (\mathbf{V}_2 - \mathbf{V}_1) \times (\mathbf{V}_3 - \mathbf{V}_2)$$

When this normal has zero length, a workstation-dependent geometric normal will be used.

Polygon With Data 3

This primitive is essentially the same as polygon 3. However, the application can explicitly specify its geometric normal in the primitive definition. If it is not specified, the normal is calculated as for polygon 3.

Triangle Strip 3

This primitive generates area geometries, each of which is a triangle defined by three consecutive vertices. When the primitive has *n* vertices, it generates *n* - 2 triangles and the i sup th triangle is defined by vertices $\mathbf{V}_i$, $\mathbf{V}_i + 1$ and $\mathbf{V}_i + 2$. If fewer than three vertices are specified, no geometries are generated. The application can specify a geometric normal for each area of the primitive definition. When the geometric normals are not defined, the geometric normal of each triangle is calculated as follows:

1. for the odd numbered triangle
   $\mathbf{N} = (\mathbf{V}_{i+1} - \mathbf{V}_i) \times (\mathbf{V}_{i+2} - \mathbf{V}_{i+1})$
2. for the even numbered triangle
   $\mathbf{N} = (\mathbf{V}_i - \mathbf{V}_{i+1}) \times (\mathbf{V}_{i+2} - \mathbf{V}_i)$

When a normal has zero length, a workstation-dependent geometric normal will be used.

Quadrilateral mesh 3

This primitive generates area geometries, each of which is a quadrilateral defined by four vertices. When the primitive has *m* x *n* vertices, it generates (*m-1*) x (*n-1*) quadrilaterals. A quadrilateral within the mesh is defined by vertices $\mathbf{V}_{i,j}$, $\mathbf{V}_{i,j+1}$, $\mathbf{V}_{i+1,j}$, and $\mathbf{V}_{i+1,j+1}$. The application can specify a geometric normal for each area of the primitive definition. When the geometric normals are not defined, the geometric normal of each quadrilateral is calculated as the normalized cross product of its diagonals as follows:

$$\mathbf{N}_{i,j} = \frac{(\mathbf{V}_{i+1,j+1} - \mathbf{V}_{i,j}) \times (\mathbf{V}_{i,j+1} - \mathbf{V}_{i+1,j})}{|(\mathbf{V}_{i+1,j+1} - \mathbf{V}_{i,j}) \times (\mathbf{V}_{i,j+1} - \mathbf{V}_{i+1,j})|}$$

Figure 73. .

where

$1 <= i <= m - 1$; $1 <= j <= n - 1$; $\mathbf{N}_{i,j}$ is the normal, $\mathbf{V}_{i,j}$ are the vertex coordinates, and *m* and *n* are the dimensions of the two-dimensional array of vertex data (*m x n*).

When a normal has zero length, a workstation-dependent geometric normal is used.

(Trimmed) Non-Uniform B-Spline Surface

In the mathematical sense, the geometric normal at a given point on a surface is defined as the cross product of two partial derivatives of the surface at the point as follows:

$$N = \frac{\delta S}{\delta u} \times \frac{\delta S}{\delta v}$$

Figure 74. .

When the surface is tessellated, a geometric normal of each area geometry will be determined as if the area is explicitly defined by a Polygon 3 primitive with vertices resulting from the tessellation or taken from the mathematical normal at some point within a corresponding parameter range. When the resulting normal has zero length, a workstation dependent geometric normal will be used.

The geometric normals defined above are transformed into World Coordinates (WC) as vertex coordinates are.

Reflectance Normal

Each point of a given area geometry has its own normal vector named a reflectance normal which is used in the lighting calculation described later.

For area geometries generated by the following primitives, reflectance normals of all points on a given area are always the same as the area's geometric normal.

Polygon 2/3
Rectangle 2
Composite Fill Area 2
Polygon Text 2/3

When the following primitives have vertex normals specified as optional vertex data, a reflectance normal of a given point is determined by interpolating vertex normals in a workstation-dependent way.

Polygon With Data 2/3
Triangle Strip 3
Quadrilateral Mesh 3

When these primitives do not have vertex normals, the reflectance normal is taken from its geometric normal and so all points on a given area geometry have the same reflectance normal.

For the following surface primitives, the reflectance normal of a given point is determined from their surface equations:

Non-Uniform B-Spline Surface
Trimmed Non-Uniform B-Spline Surface

When these primitives are tessellated, mathematical normals at vertices resulting from the tessellation process are calculated precisely from their surface equations. A reflectance normal at other points will be obtained by interpolating these normals.

In the graPHIGS API, all reflectance normals are transformed using the same method that is used for geometric normals.

Modeling Clipping

Overview

The modeling clipping function provides a way to clip geometric entities in world coordinates, just after the modeling transformation and before any other rendering pipeline operations.

The graPHIGS API maintains current model clipping volume used to ″clip″ geometric entities during traversal. Any portion of element geometry outside the clipping volume is discarded. Modeling clipping operates only on primitive elements and has no effect on other elements, such as Test Extent 2/3 **(GPTEX2** and **GPTEX3**).

Modeling Clipping is supported by the following subroutines:

Set Modeling Clipping Volume 2 **(GPMCV2)**
Set Modeling Clipping Volume 3 **(GPMCV3)**
Restore Modeling Clipping Volume **(GPRMCV)**
Set Modeling Clipping Indicator **(GPMCI)**

Use the Inquire Workstation Description **(GPQWDT)** subroutine to determine whether modeling clipping is supported on your workstation. On workstations that do not support modeling clipping, the maximum number of half-spaces (*maxhs*) and the number of available clipping operators (*loper*) are set to zero.

Setting the Clipping Volume

The clipping volume in world coordinates is set by the Set Modeling Clipping Volume 2/3 **(GPMCV2)** and **(GPMCV3)** elements with the following parameters:

Operator

This parameter is an integer specifying an operation to be performed between the current clipping volume and that which is specified in this element. The Modeling Clipping Operator is defined as:

1= REPLACE_VOLUME
Set the current clipping volume to that specified in this element.

2= INTERSECT_VOLUME
Set the current clipping volume to the intersection (logical AND) of the current clipping volume and that specified in this element.

The resulting clipping volume becomes the current modeling clipping volume and is not affected by subsequent modeling transformation elements encountered during traversal.

List of Half Spaces

This parameter specifies a clipping volume in modeling coordinates. Each entry of this list contains a point and a direction vector (normal). The point and normal are transformed by the current modeling transformation into world coordinates, where the transformed point is on a clipping plane and the transformed normal points into the adjoining unbound region called *half-space* Each of the half spaces in the list are intersected to form an *acceptance region* Element geometries outside of the acceptance region are rejected or ″clipped″.

Number of Clipping Planes

This parameter specifies the number of clipping planes (half spaces). Theoretically the graPHIGS API places no limit on the number of clipping planes that can be specified in the Set Modeling Clipping Volume 2 and 3 elements, and no limit to what can be implemented in any particular workstation. However, it may not be necessary or practical to specify an extremely large number. Generally

optimum results are achieved with six or fewer clipping planes specified at a time. The most commonly discussed uses of clipping planes involve 1 plane (slicing), 2 planes (limiting) or 6 planes (bounding).

Therefore, the limit of the number of clipping planes available is workstation dependent; your application should inquire the number of available clipping planes using **GPQWDT**.

Restoring the Current Clipping Volume to the Inherited Setting

The Restore Modeling Clipping Volume **(GPRMCV)** subroutine is also available to change the current modeling clipping volume. It takes no parameters and restores the current clipping volume to the one inherited by that structure, i.e., the clipping volume when the current structure was invoked.

Turning Modeling Clipping On and Off

Modeling clipping (with the current clipping volume) is activated or deactivated by the Set Modeling Clipping Indicator **(GPMCI)** element which takes `1=CLIP` or `2=NOCLIP` as its parameter.

Note: Traversal defaults for the modeling clipping indicator and modeling clipping volume are `2=NOCLIP` and all world coordinate space *(i.e., 0 half spaces)*, respectively. This means that even if the current modeling clipping indicator is `1=CLIP`, no geometric entities are clipped if the modeling clipping volume is all of world coordinate space.

Modeling Clipping of Point, Curve, and Surface Primitives

Conceptually, the modeling clip for primitives should be performed by using their mathematical definition. The graPHIGS API defines three classes of geometry for the supported primitives:

- **Point primitives**

  These include the primitives for annotation text, markers, and the pixel primitives. A point primitive is clipped only if its associated point is clipped. If the point is not clipped, then the primitive is expanded to co-planar geometries parallel to the *xy*-plane of NPC. It then may be clipped by the view clip and/or workstation clip.
- **Curve primitives**

  These include the primitives for lines, arcs, geometric text, and curves. A curve primitive is clipped using its mathematical definition without considering its width.
- **Surface primitives**

  These include the area-defining primitives for polygons, surfaces, triangle strip, quadrilateral mesh, and composite fill area. Clipping of a surface primitive is achieved by performing a modeling clip of the boundary of the primitive.

Modeling clipping generates new vertices of output primitives at the intersection of the clipping planes and the primitive's boundary. If color, data, alpha (transparency coefficient), or vertex normals are associated with the original vertices, then new values are created for the new vertices by interpolation.

Face and Edge Culling

Polygon Culling

Polygon culling is a process to determine whether a given area geometry should be visualized or not based on its relation to its viewer. In the graPHIGS API, polygon culling is performed in World Coordinates (WC) based on its geometric normal and direction vector from the area to the viewer.

Let `PRP` and `COW` be the Projection Reference Point and the Center of Window in VRC. Transforming these points by the inverse of the view matrix, you get `PRP'` and `COW'` in WC. The direction vector in a parallel projection is a vector from `COW'` to `PRP'`. The direction vector in a perspective view is a vector from a point on the area geometry to `PRP'`. Note that this assumes the view matrix is a normal 3D transformation, that

is, represented by a 4 x 4 matrix with (0,0,0,1) elements in it the fourth column. Unless such a matrix is irregular, the inverse transformation of the PRP and COW are well defined. When the matrix has no inverse, polygon culling and polygon edge culling (see later) are not performed at all.

Let E_d be a unit direction vector to the viewer and Nbe the geometric normal of the area geometry. When a dot product
has a negative value, the area is back facing to the viewer. Otherwise, it is front facing to the viewer.

$E_d \cdot N$

Figure 75. .

The application can control whether each area geometry (including its boundaries/edges) should be visualized or not by the Polygon Culling Mode attribute set by the Set Polygon Culling Mode **(GPPGC)** structure element. The element takes one of the following values:

1=NONE (default)
No polygon culling is performed. All area geometries are visualized independent of their relation to the viewer.

2=BACK
Area geometries which are back facing are not visualized.

3=FRONT
Area geometries which are front facing are not visualized.

Polygon Culling of Edges

By this definition, polygon culling is applied to area geometries, which include both interiors and edges. When an edge is shared by two area geometries there is an ambiguity whether the edge should be visualized or not if one area is back facing and another is front facing. Polygon culling of edges is defined conceptually as follows:

- A shared edge is visualized when either of the two adjacent areas is determined to be visualized.
- When both areas are visualized, the shared edge is drawn only once. However, the actual processing of the shared edge is workstation-dependent.

Polyhedron Edge Culling

The polyhedron edge primitive generates a line geometry. Its primary use is to define a line segment which is an edge of a polyhedron, or the intersection of two planar area geometries. Each polyhedron edge has two normals representing geometric normals of the adjacent areas. Each area will be front facing or back facing as defined for area geometries. The application can control whether the polyhedron edge should be visualized or not by the Polyhedron Edge Culling Mode attribute. The Set Polyhedron Edge Culling Mode **(GPPHEC)** structure element takes one of the following values:

1=NONE (default)
All polyhedron edges are always visualized independent of their normals.

2=BOTH_BACK
A polyhedron edge with two areas both back facing is not visualized.

3=BOTH_FRONT
A polyhedron edge with two areas both front facing is not visualized.

4=BOTH_BACK_OR_BOTH_FRONT
A polyhedron edge with two areas facing the same direction is not visualized (edges are displayed if the areas face different directions)

5=BACK_AND_FRONT
A polyhedron edge with two areas facing opposite directions is not visualized.

6=LEAST_ONE_BACK
A polyhedron edge with any area back facing is not visualized.

7=LEAST_ONE_FRONT
A polyhedron edge with any area front facing is not visualized.

Other combinations are meaningful but are not currently supported.

Lighting, Shading, and Color Selection

Overview

Lighting and shading simulates the effect of light sources shining on area primitives.

Color Selection

Prior to lighting or shading, a *color selection* process determines the source of color used with area primitives. Depending on the attributes applied to the primitives, the selected colors may be the highlighted color, a pattern of colors, or the color values determined by *data mapping.* The color selection process is described below.

Lighting

The lighting calculations modify the colors of area primitives based on values assigned to light sources (*illumination*) and their interaction with the primitive (*reflection*). (See Lighting.)

Shading

The shading calculation (*interpolation*) defines how the interior of the primitive is filled. For example, the result of lighting calculations performed at each vertex is interpolated across the interior of the primitive. (See Shading.)

Color Selection

- If you specified highlighting for an area primitive using a Set Highlight Color subroutine (**GPHLCD** or **GPHLCI**), the highlight color is used as the color of the primitive, and no further color selection is done.
- If you specified the Interior Style as `3=PATTERN` using the Set Interior Style (**GPIS**) subroutine, the interior is filled with the pattern.
- If you specified the Interior Style as `1=HOLLOW`, `2=SOLID`, or `4=HATCH`, the Data Mapping Method determines the color as follows:
  - If the Data Mapping Method is `1=DM_METHOD_COLOR`, then one of the color attributes of the primitive is used:

    If vertex colors are present, then the vertex color is used.

    If no vertex colors are present, then the facet color is used, if present. (This applies only to those primitives that support a facet color.)

    If neither vertex nor facet colors are present, then the current interior color is used. The current color is determined by resolving the interior color Attribute Source Flag (ASF) to select the appropriate (back) interior color from the current attribute or the current interior bundle table entry.
  - If the Data Mapping Method is `-1=IMAGE_ARRAY`, `2=SINGLE_VALUE_UNIFORM`, or `4=BI_VALUE_UNIFORM`, then the Data Mapping Method determines the color used.

The Data Mapping Method uses data values specified at the vertices of certain primitives to determine the colors to be used with the primitives. These advanced primitives include:

Polygon 2 with Data **(GPPGD2)**
Polygon 3 with Data **(GPPGD3)**
Triangle Strip 3 **(GPTS3)**
Quadrilateral Mesh 3 **(GPQM3)**

Refer to Texture/Data Mapping for information about using data mapping.

Lighting

In the graPHIGS API, lighting is treated as an individual attribute for specifying or modifying the color of an area geometry. Lighting calculations simulate the effects of light sources illuminating an object, and are performed only on area geometries.

Lighting effects on a given point are calculated in two steps.

Step One: *Light Source Calculation*
This step involves calculating the intensity of the light reaching the surface which is dependent on the Light Source Type parameter.

Step Two: *Reflectance Calculation*
This step involves determining the amount of light reflected from the surface which is simulated by computing three types of reflection: *ambient, diffuse,* and *specular.*

You can control the lighting calculation and the resulting quality of the displayed color by using the Set Lighting Calculation Mode **(GPLMO)** subroutine (See Set Lighting Calculation Mode) as well as the Set Reflectance Model **(GPRMO)** and the Set Back Reflectance Model **(GPBRMO)** subroutines. (See Step Two: Reflectance Calculations.) In addition, you can control shadow effects by using the Set Face Lighting Method **(GPFLM)** attribute subroutine. (See Face-dependent Lighting Effects.)

All lighting calculations in the graPHIGS API are performed using World Coordinates (WC). All vertices and normal vectors are transformed by the modeling transformation and clipped to the modeling volume. Light source information specified in the light source table contains only World Coordinate values and are not transformed by modeling transformations.

Step One: Light Source Calculation

Light source characteristics are stored in a workstation table. These characteristics can be changed with the Set Light Source Representation **(GPLSR)** subroutine. The four Light Source Types currently defined are

```
1=AMBIENT
2=DIRECTIONAL
3=POSITIONAL
4=SPOT
```

Light sources can be activated and deactivated through the Set Light Source State structure element created with the Set Light Source State **(GPLSS)** subroutine.

The Light Source Equation

For each active light source, the intensity vector ($I = (I_1, I_2, I_3)$) of the incoming ray is calculated based on the light source type and color ($L = (L_1, L_2, L_3)$) as illustrated the following figures:

AMBIENT light

The following figure shows an ambient light:

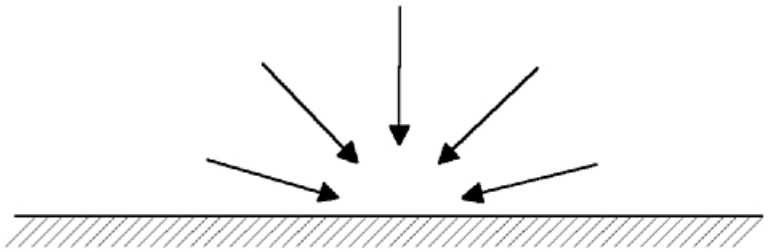


Figure 76. Ambient Light Source Definition. This illustration depicts ambient light. Ambient light comes from all directions to light the surface; there is no one-source.

The intensity of the incident light (*I*) is equal to the light source color (L) as defined below:

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \\ L_3 \end{bmatrix}$$

Figure 77. Ambient Light Equation. This illustration shows the following equation: $[I_1\ I_2\ I_3] = [L_1\ L_2\ L_3]$

DIRECTIONAL light

With a directional light source, all rays are parallel and the intensity of the light is assumed to ***not*** diminish with distance as shown in the following figure:

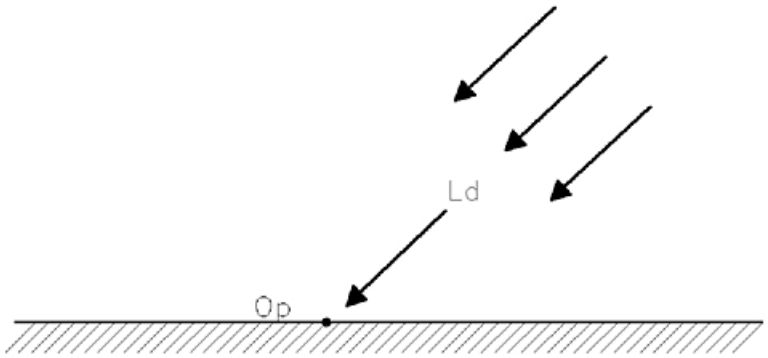


Figure 78. Directional Light Source Definition. This illustration shows how directional light effects a surface. Directional light comes from a source, and the light rays from this source are parallel. In this illustration the rays are traveling from the top right corner to the surface. The light source direction is labeled as L_d, and a point on the surface is labeled as O_p.

Therefore, the intensity vector **I** of the light reaching a point on a surface is given by the following expression:

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \\ L_3 \end{bmatrix}$$

Figure 79. Directional Light Equation. This illustration shows the following equation: $[I_1\ I_2\ I_3] = [L_1\ L_2\ L_3]$

POSITIONAL light

A positional light source illuminates a point on a surface as shown in the following figure:

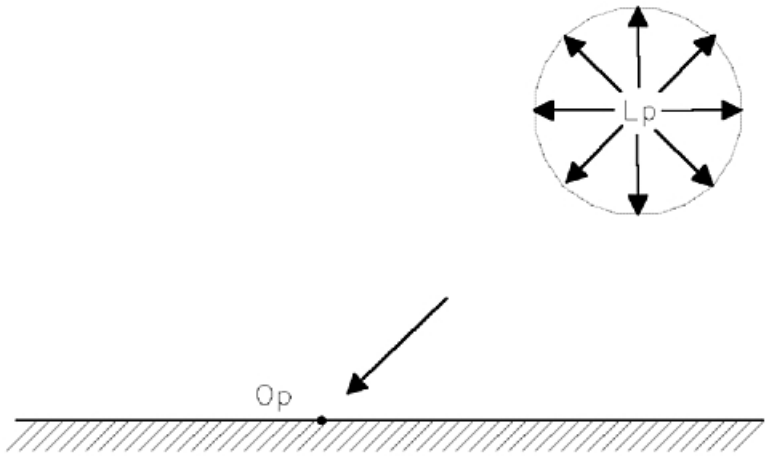


Figure 80. Positional Light Source Definition. This illustration depicts a positional light located in the upper right hand corner. The location of the light source is labeled as L_p and the light rays emitting from this source travel outward in all directions, much like the rays from the sun. A point on the surface is labeled as O_p.

The illumination is calculated based on two light source attenuation coefficents (a_1 and a_2) and the distance of the point ($\mathbf{O}_p$) relative to the location of the light source ($\mathbf{L}_p$) as follows:

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \\ L_3 \end{bmatrix} \frac{1}{a_1 + a_2 \, | O_p - L_p |}$$

Figure 81. Positional Light Equation. This illustration shows the following equation: $[I_1\ I_2\ I_3] = [L_1\ L_2\ L_3]\ (1 / ((a_1 + a_2)(|O_p - L_p|))$

SPOT light

A spot light illuminates a surface point as shown in the following figure:

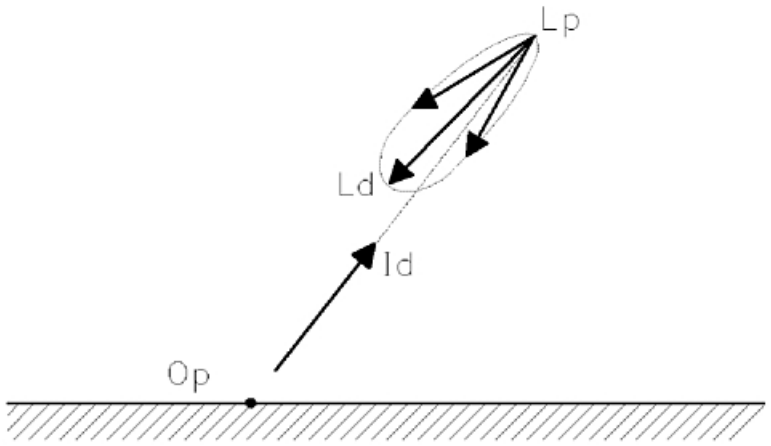


Figure 82. Spot Light Source Definition. This illustration depicts the various components of a spot light definition. The location of the spot light is denoted by L_p, and its source direction is denoted as L_d. A point on the surface which falls within the light cone is labeled as O_p. This point is illuminated; the illumination direction is depicted as an arrow pointing back to the spot light source and is denoted by I_d.

The calculation of the illuminance is based on two light source attenuation coefficents (a_1 and a_2), a light source concentration exponent (c_a), the distance from the point ($\mathbf{O}_p$) to the light ($\mathbf{L}_p$), the light source direction ($\mathbf{L}_d$), and the spread angle (theta) of the light rays. The spread angle is the angle between the light source direction and the edge of the light cone; it defines the radius of the cone of light. If the illuminated point falls within the cone defined by the spread angle theta then the illumination is calculated as follows:

If the illuminated point falls outside of the cone defined by theta then the components of the

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \\ L_3 \end{bmatrix} \frac{\left| \frac{L_d \bullet (O_p - L_p)}{|O_p - L_p|} \right|^{c_e}}{a_1 + a_2 |O_p - L_p|}$$

Figure 83. Spot Light Equation. This illustration shows the equation for calculating the illuminance of spot light is as follows: $[I_1\ I_2\ I_3] = ([L_1\ L_2\ L_3](((l(l_d$ dot product $(O_p - L_p))/(|O_p - L_p|)|_{ce})/(a_1 + a_2|O_p - L_p|))$.

illumination vector **I** are all set to zero.

Step Two: Reflectance Calculations

Step two in the lighting process involves the calculation of color values from the light source values and the primitive attribute values, such as the surface properties and specular color. The Set Reflectance Model **(GPRMO)** and the Set Back Reflectance Model **(GPBRMO)** subroutines specify which terms of the calculation are performed.

The reflectance calculations are conceptually performed at each position of the interior of the primitive (except when the Interior Style is `5=EMPTY`, which displays no interior.) However, this calculation at every pixel may not be performed due to the Interior Shading Method selected; see Shading.

The defined reflectance models are:

1=REFLECTANCE_NONE

This is the traversal default. No reflectance calculation is performed. (The same effect can be achieved also by setting appropriate reflection coefficients to 0.0, causing the terms to provide no effect.)

2=AMB
Only the ambient term of the reflectance calculation is computed.

3=AMB_DIFF
The ambient and diffuse terms of the reflectance calculation are computed.

4=AMB_DIFF_SPEC
All terms (ambient, diffuse, and specular) of the reflectance calculation are computed.

Notes:

1. Typically, an application uses the models `1=REFLECTANCE_NONE` and `4=AMB_DIFF_SPEC`
2. Lighting and shading do not apply to edges.
3. A specular exponent of 0.0 (the default) will result in no specular effect.

The following figure illustrates the second step in the lighting calculation process:

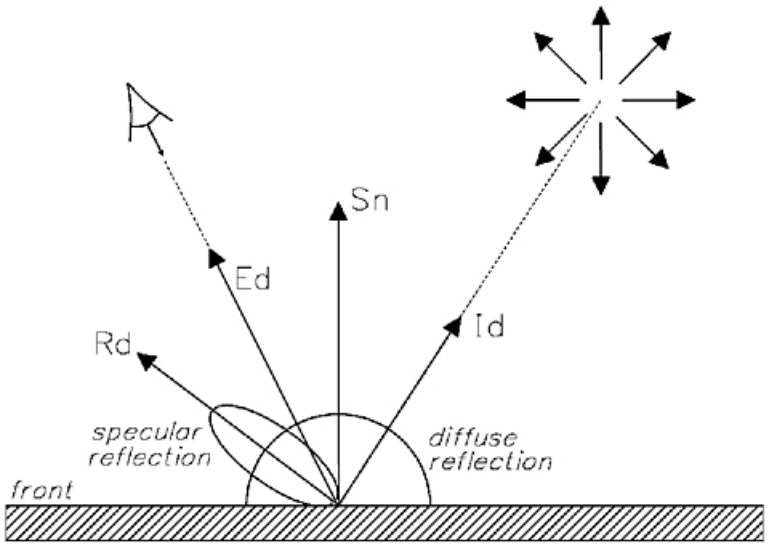


Figure 84. Reflected Light Definition. This illustration depicts the second step in the lighting calculation process. There is a light source defined in the upper right hand corner that appears as a postional light. I_d denotes the direction of the incoming light. The unit reflectance normal of the illuminated point is denoted by S_n. In this illustration, this normal is reflecting upwards from the surface in a northern direction. An eye (observer) is depicted in the upper left corner and the direction vector from the illuminated point to the eye is denoted by E_d. The ray of light that is emitted from the light source causes a reflection off of the surface. The unit direction vector of this reflected ray of light is denoted as R_d. In a diffuse reflection, light is scattered in all directions from a point on the surface. In a specular reflection, light is concentrated around the mirror direction R_d.

The Reflectance Equation

The light reflected from an area geometry is the summation of the light reflected from each individual light source. For ambient light sources, the j^{th} component of the reflected light is equal to the summation of the following quantity over all active ambient light sources:

$$a_c \quad I_j \quad D_j \quad \text{for} \quad j = 1,2,3$$

where:

a_c is the ambient reflection coefficient of the surface

I_j is the j^{th} component of the incident light

D_j is the j^{th} component of the surface's diffuse color.

For all other light source types, the reflected light is equal to the sum of the diffuse and specular terms.

The diffuse term is defined to be the summation of the following quantity over all active light sources (other than ambient):

$$d_c \quad D_j \quad \mathrm{I}_j \quad (\mathbf{S}_n \cdot \mathbf{I}_d) \quad \text{for } j = 1,2,3$$

Figure 85.

where:

d_c is the diffuse reflection coefficient of the surface

D_j is the j^{th} diffuse color component of the surface

I_j is the intensity of the incoming ray of light

$\mathbf{I}_d$ is the direction of the incoming ray of light

$\mathbf{S}_n$ is the unit reflectance normal of the illuminated point.

When the dot product

$$(\mathbf{S}_n \cdot \mathbf{I}_d)$$

is less than zero, it is treated as zero and so the light source does not contribute to this term.

The specular term is defined to be the summation of the following quantity over all active light sources (other than ambient):

$$S_c \quad S_j \quad I_j \quad (\mathbf{E}_d \cdot \mathbf{R}_d)^s e \quad \text{for } j = 1,2,3$$

where:

s_c is the specular reflection coefficient of the surface

s_e is the specular reflection exponent of the surface

S_j is the j^{th} specular color component of the surface

I_j is the j^{th} intensity component of the incoming ray of light

$\mathbf{E}_d$ is the unit direction vector from the illuminated point to the observer

$\mathbf{R}_d$ is the unit direction vector of the reflected ray of light calculated as follows:

$$\mathbf{R}_d = 2\ (\mathbf{S}_n \cdot \mathbf{I}_d)\ \mathbf{S}_n - \mathbf{I}_d$$

When either the dot product

$$(\mathbf{E}_d \cdot \mathbf{R}_d)$$

or the dot product

$(\mathbf{S}_n \cdot \mathbf{I}_d)$

is less than zero, it is treated as zero and so the light source does not contribute to this term.

Face-dependent Lighting Effects

The face-dependent lighting method is based on the real-world effect that causes shadows. If a light is on one side of an opaque object and the viewer is on the other side, then the viewer would not see any illumination from the light. The viewer's side would be dark, or in the shadows, or at least not as bright as the side facing the light.

The lighting process creates this effect by first determining where the lights and viewer are, then changing some of the values in the lighting calculation if viewer and lights are on opposite sides.

Use the Set Face Lighting Method **(GPFLM)** attribute subroutine to enable this effect through `2=FACE_DEPENDENT`. (By default, the effect is off; i.e., lighting is `1=FACE_INDEPENDENT` since the relationship of lights and viewer to the area is ignored).

When the Face Lighting Method is `2=FACE_DEPENDENT`, then:

- **Position Relationships**

  The relationships of the positions of the viewer and the lights to the area is determined by calculations of various vector cross-products; the results define the orientation of the position to the face.

  – ***Viewer Position***

    The position of the viewer is used with the geometric normal to determine the viewer orientation to the front face of the area. An object is front-facing if the geometric normal is pointing to the viewer.

  – ***Light Position***

    The Light Source Representation table defines a position for each light that enters into the lighting calculations. You use this light position with the geometric normal to determine the orientation of the light to the front face of the area. The light is then determined to be on the front or back side.

    Note: The use of the geometric normal to determine the orientation of the light relative to the front face of the area may have a performance impact. For this reason, an approximation may be performed using the vertex normal rather than the geometric normal. The two normals are close in value for most situations, so this approximation yields good results without degrading performance.

- **Lighting Calculations**

  From the above processing, the position relationships of the viewer and light to the area are determined. If the light and the viewer are on opposite sides of the area, then the diffuse and specular terms of the lighting processing are suppressed (i.e. the values are set to 0). Only these two terms of the lighting calculations are affected since they are the only terms whose reflection is dependent on the position of the light source.

The effects of face-dependent lighting can be summarized based on the relationships of the light and the viewer to the front and back faces of the area.

- *Light and viewer both on front side*

  All lighting calculations proceed as defined.

- *Light on front side, viewer on back side*

  Diffuse and specular terms are both set to 0. All other lighting calculations proceed as defined.

- *Light on back side, viewer on front side*

  Diffuse and specular terms are both set to 0. All other lighting calculations proceed as defined.

- *Light and viewer both on back side*

Back-side vertex normals are calculated by inverting the vertex normals. These back-side vertex normals are then used in the lighting calculations.

Performance Impact of Implicit Normals

The geometric normal can explicitly be provided in some primitives or can be implicitly generated by calculations performed by the graPHIGS API. Because implicit normals must be generated on every traversal, a performance impact may result. Therefore, your application programs should provide explicit normals for lighting and Hidden Line/Hidden Surface Removal (HLHSR) processing.

Shading

The Set Interior Shading Method **(GPISM** and **GPBISM**) subroutines define how to fill the interior of area primitives. Except for `1=SHADING_NONE`, the shading methods use interpolation to perform the fill process. Interpolation is the process of calculating data at intermediate points along at primitive's geometry from data at the primitive's vertices. It results in averaging the data across a primitive, typically from data defined at each vertex.

The Interior Shading Method options are:

1=SHADING_NONE

This method specifies that no shading is to be done.

This shading method is traditionally known as ″flat″ shading because the same color is used everywhere on the facet.

A single color value is used everywhere for the interior color of the primitive. The method for determining the color used is described in Color Selection. If lighting calculations are enabled, then a lighting calculation is performed on the color value.

2=SHADING_COLOR

This method is the default. It specifies that shading is done by the interpolation of color values across the interior of the primitive.

This shading method is traditionally known as ″Gouraud″ shading, and appears as colors that blend from one vertex to another.

If lighting calculations are enabled, then a lighting calculation is performed on the color value at each vertex. The method for determining the colors used is described in Color Selection. The resulting colors are linearly interpolated across the primitive.

If `2=SHADING_COLOR` is used but no vertex color values are specified, the Interior Shading Method defaults to `1=SHADING_NONE`.

3=SHADING_DATA

This method specifies that shading is done by the interpolation of the data mapping data values before they are converted into colors.

If lighting is enabled, then a lighting calculation is performed on each vertex, independent of the data and colors. The data values are then linearly interpolated across the primitive along with these independent lighting calculations. Next, the interpolated data values are used to determine the color values of the interior of the primitive. Finally, these interior colors are combined with the interpolated lighting calculations to produce a lit, *texture mapped* interior.

If `3=SHADING_DATA` is used with the data (*texture*) mapping method (`1=DM_METHOD_COLOR`) and no vertex colors are specified, the method defaults to `1=SHADING_NONE`

When Color Values Are Used As Data Values

There are instances when data values may not be used during Color Selection. Specifically, the

highlighting color or pattern may be used, or the data mapping index may select 1=DM_METHOD_COLOR (See Color Selection.) In such instances, the colors themselves are used as data mapping *data values*. Processing occurs as follows:

If lighting is enabled, then:

- the lighting calculations are performed independently from the colors
- the colors and lighting calculations are interpolated independently
- the interpolated values are combined at the very end to determine the interior color values
- the resulting appearance is very similar to that of 2=SHADING_COLOR

Set Lighting Calculation Mode

The Set Reflectance Model **(GPRMO)** and Set Interior Shading Method **(GPISM** and **GPBISM**) subroutines give you more flexibility over the graPHIGS API lighting and shading controls. However, for compatibility, the Set Lighting Calculation Mode **(GPLMO)** is also available.

Conceptually, the lighting calculations described here are performed for every point on an area geometry.

However, to utilize workstation capabilities and obtain the required performance, your application can control the lighting process through the Set Lighting Calculation Mode **(GPLMO)** subroutine which creates a structure element with one of the following values:

1=NONE

No lighting calculation is performed in this mode. Colors determined by the color selection stage are passed to the depth cueing stage with no modification. This mode is the traversal default of the graPHIGS API.

Note: This mode does not necessarily mean that area geometries are displayed with a constant color. Depending on your workstation, interpolation of vertex colors and depth cueing may still be applied to the primitive.

2=PER_AREA (at least one for an area geometry)

At least one lighting calculation is performed for each area. The resultant color is assigned to all points of the area. The point where the lighting is performed is workstation dependent. Possible alternatives are:

- the first vertex
- the center of gravity
- an arbitrary vertex.

This mode does not necessarily mean the area geometry is displayed with a constant color and does not inhibit performing more lighting calculations. For polygons with vertex colors, lighting calculations may be performed for every vertex even in this mode.

3=PER_VERTEX (at least for every vertex)

Lighting calculations should be performed at least for all vertices of the area geometry. Colors of other points will be obtained by interpolating resultant colors at vertices.

Hidden Line/Hidden Surface Removal (HLHSR)

Hidden Line / Hidden Surface Removal (HLHSR) is a process used to enhance the display of 3-dimensional objects: by controlling how objects that are closer to the viewer are rendered with objects that are farther from the viewer, the application can achieve certain desired effects. For example, the application can cause hidden parts to not appear on the display, or it can display the hidden parts in a special way that indicates to the user that the parts are hidden.

HLHSR is typically implemented using a Z-buffer. A Z-buffer is storage that holds the Z-value of a pixel: this value is used to compare with other pixel Z-values that are generated during the rendering process. The comparison determines whether the frame buffer and/or the Z-buffer are to be updated with the new pixel or are to remain unchanged.

For example, consider a common HLHSR process of removing hidden surfaces. As each pixel of a surface primitive is generated, its Z-coordinate value is compared to the Z-coordinate value in the Z-buffer (if a Z-value already exists). If the new Z-value is closer to the viewer, then the new pixel value replaces the current pixel value in the frame buffer and the Z-buffer is updated with the Z-value of the new pixel. If the new Z-value is farther from the viewer, then the new pixel value does not replace the current pixel value. The effect is that closer objects overwrite objects that are farther away, hiding them. Note that both the frame buffer and the Z-buffer are updated if the comparison indicates that the new pixel should replace the existing pixel.

The Set HLHSR Identifier element is used to specify how each geometric entity is to be processed in the HLHSR process. The following table lists the HLHSR identifiers and summarizes the effect of the frame buffer and Z-buffer.

Table 3. HLHSR Processing

	Frame buffer	**Z-buffer**
1: VISUALIZE_IF_NOT_HIDDEN	$Z_{prim} >= Z_{buf}$	$Z_{prim} >= Z_{buf}$
2: VISUALIZE_IF_HIDDEN	$Z_{prim} < Z_{buf}$	Never
3: VISUALIZE_ALWAYS	Always	Always
4: NOT_VISUALIZE	Never	$Z_{prim} >= Z_{buf}$
5: FACE_DEPENDENT_VISUALIZATION		
Front-facing Areas	$Z_{prim} >= Z_{buf}$	$Z_{prim} >= Z_{buf}$
Back-facing Areas	$Z_{prim} > Z_{buf}$	$Z_{prim} > Z_{buf}$
6: NO_UPDATE	Never	Never
7: GREATER_THAN	$Z_{prim} > Z_{buf}$	$Z_{prim} > Z_{buf}$
8: EQUAL_TO	$Z_{prim} = Z_{buf}$	$Z_{prim} = Z_{buf}$
9: LESS_THAN	$Z_{prim} < Z_{buf}$	$Z_{prim} < Z_{buf}$
10: NOT_EQUAL	$Z_{prim} <> Z_{buf}$	$Z_{prim} <> Z_{buf}$
11: LESS_THAN_OR_EQUAL_TO	$Z_{prim} <= Z_{buf}$	$Z_{prim} <= Z_{buf}$

Identifiers 1-5 define special HLHSR effects:

1=VISUALIZE_IF_NOT_HIDDEN
If the Z-value of the generated pixel is closer to the viewer or the same distance as the Z-buffer value, then both the Z-buffer and the frame buffer are updated with the generated pixel value. This causes closer objects to overwrite farther objects.

2=VISUALIZE_IF_HIDDEN
If the Z-value of the generated pixel is farther from the viewer than the Z-buffer value, then only the frame buffer is updated with the generated pixel value, and the Z-buffer is not updated. This causes hidden objects to be displayed, but they do not update the Z-buffer since they are not closer to the viewer.

3=VISUALIZE_ALWAYS
The frame buffer is updated with the generated pixel value. In addition, if the Z-value of the generated pixel is closer to the viewer or the same distance as the Z-buffer value, then the Z-buffer is updated with the generated Z-value. This causes all objects to be displayed, but they do not update the Z-buffer unless they are closer to the viewer.

4=NOT_VISUALIZE

If the Z-value of the pixel is closer to the viewer or the same distance as the generated Z-buffer value, then the Z-buffer is updated with the generated pixel Z-value. The frame buffer is not updated. This causes no objects to be written to the frame buffer, and objects do not update the Z-buffer unless they are closer to the viewer.

5=FACE_DEPENDENT_VISUALIZATION

For *back-facing areas*, the frame buffer and Z-buffer are updated if $Z_{prim} > Z_{buf}$.

For *all other entities* (front-facing areas and non-area defining entities), the frame buffer and Z-buffer are updated if $Z_{prim} >= Z_{buf}$.

This definition is useful for the processing of coincident geometries. Consider the silhouette edges of solid objects, where the front-facing geometry meets the back-facing geometry, and assume that a front-facing area is already rendered in the frame buffer and Z-buffer and a back-facing area is then processed. Where the two areas have equal Z-values (for example, at the shared edge), then the front-facing pixels are not replaced by the back-facing pixels, (even though the back-facing area is at equal Z-value and rendered later). Otherwise, back-facing colors would ″pop″ out onto the front-facing area at the shared edge. The effect of this definition is to give preference to front-facing primitives over back-facing primitives when the Z-values are equal.

The front face of an area is defined by the geometric normal. If the geometric normal is pointing to the viewer, then the object is front-facing. The geometric normal can be explicitly provided in some primitives, or can be implicitly generated by calculations. Since implicit normals must be generated on every traversal, a performance impact may result. Therefore, your application programs should provide explicit normals for lighting and HLHSR processing.

The overall effect of HLHSR processing is that, except for `VISUALIZE_IF_HIDDEN` processing, closer objects are in the Z-buffer, and the HLHSR processing controls whether these are the objects in the frame buffer (i.e. whether these objects are displayed).

Generation of the Z-value of a pixel is a process that depends on the primitive rendered, its transformation state, and the hardware that is used by the workstation. Conceptually, each unique pixel represents a single value out of a range of potential Z-values that the primitive can generate. This value is dependent on the method of generation that the hardware uses. For example, line primitives may generate different Z-values than a polygon primitive at the same pixel, even though the primitives use the same coordinates and transformations (and thus the pixel is conceptually the same). In large part, these differences are due to the way that the workstation hardware implements the transformation facilities and how the calculated floating-point values are converted to integer values that the hardware may use. Due to these differences, inaccuracies are possible that may give unpredictable results in some situations.

HLHSR processing of surfaces has an additional consideration. In general, a surface is tesselated into polygons for processing. This tessellation is an approximation of the conceptual surface, and the resulting polygons will not correspond at each pixel as the theoretical surface would. Thus, it is unpredictable how two close surfaces will display during HLHSR processing as a result of tesselation.

Since the hardware that implements HLHSR may be different on different workstations, see *The graPHIGS Programming Interface: Technical Reference* for any restrictions on HLHSR processing.

In the graPHIGS API products, the term “hidden or not hidden” is defined in the NPC of each view. A geometric entity in one view does not affect the HLHSR process of other views. Whether a given geometric entity is hidden or not is determined by whether there is any other geometric entity that has the same Normalized Projection Coordinates (NPC) X and Y values, but larger NPC Z coordinates. If there is such an entity, then the given entity is hidden. Otherwise it is not hidden.

Whether the HLHSR process should be performed or not and/or how the HLHSR checking should be performed is controlled by the HLHSR Mode of a view. The HLHSR Mode is controlled using the Set Extended View Representation **(GPXVR)** subroutine, which takes one of the following values:

1=OFF
In this mode, no HLHSR processing is performed. The HLHSR identifier attribute is ignored and all geometric entities are always visualized.

2=ON_THE_FLY
In this mode, the HLHSR checking is performed on the fly, i.e. when a geometric entity is processed. Whether part of the geometry is hidden or not is determined by geometric entities that have already been processed within the view. This mode corresponds to the Z-buffer technique.

HLHSR of Glyphs, Lines and Edges

Conceptually, HLHSR for these geometric entities should be performed by treating them in their mathematical sense. For example, whether an annotation text primitive is hidden should be determined only by whether the annotated reference point is hidden. When the point is hidden, the entire annotation text will be considered as hidden. However, the effects of HLHSR on these geometric entities is workstation-dependent.

Annotation Text and Marker HLHSR Procopt

If you desire not to have annotation text or markers hidden even if the reference point is hidden, use the HLHSR coordinate system processing option (PNTHLHSR). This procopt gives you the option of specifying HLHSR processing in Device Coordinates (DC) or Viewing Coordinates (VC) for annotation text and markers. Specify `2=DEVICE_COORDINATES` in PNTHLHSR when you want annotation text or marker primitives processed on a per-pixel basis in DC with the effect that text and marker strokes are z-buffered, not their reference point. Specify `1=VIEWING_COORDINATES` when you want HLHSR processing of annotation text or marker primitives based on their reference point only. For a general description of the format of this procopt, see *The graPHIGS Programming Interface: Technical Reference*.

HLHSR of Coincident Geometries

Several objects can occupy (or be intended to occupy) the same coordinate space during HLHSR processing. The resulting display depends on the types of primitives, the transformation of the primitives, and on the HLHSR implementation in the workstation.

Although two different objects may conceptually occupy the same coordinates (in the mathematical sense), the effects of floating-point inaccuracies may make the display unpredicatable in some cases. A Z-buffer in a workstation may be implemented in hardware using integer values. The floating-point Z-coordinate values must be converted to this integer format. Two floating-point coordinates that differ slightly in value may be represented by the same integer in the Z-buffer. These primitives are coincident during the HLSHR processing, and certain situations may cause unintended results to appear on the display. Likewise, two primitives that are intended to be coincident may not be when the transformed floating-point coordinates are used for HLHSR processing as they may be represented by slightly different integer values in the Z-buffer. In addition to the floating-point-to-integer conversions, different primitives may generate differing Z-values for the same pixel due to the method of generation of the pixel values from the input coordinates. For example, area-fill methods.

Chapter 17. Manipulating Color and Frame Buffers

This chapter contains the following sections:

- Color Definition
- Rendering
- Color Quantization
- Default Color Processing Configurations
- Color Processing Examples
- Frame Buffer Manipulations

Certain capabilities are available only on specific workstations. Therefore, your application should use the inquiry subroutines discussed throughout this chapter to make your application portable to other hardware platforms.

Color Definition

There are two ways to specify colors in the graPHIGS API: indexed and direct. When using *indexed color*, your application defines colors as indexes into a workstation's *rendering* color table. Each entry in a rendering color table is a color vector which can be set using the Set Extended Color Representation **(GPXCR)** subroutine. A color vector is defined as three floating-point numbers ranging from zero (0.0) to one (1.0), representing color components in the current workstation color model. Use the Set Color Model **(GPCML)** subroutine to set the workstation's color model. The supported color models are:

- `1=RGB` (Red, Green, Blue)
- `2=HSV` (Hue, Saturation, Value)
- `3=CMY` (Cyan, Magenta, Yellow)
- `4=CIELUV` (Commission Internationale de l'Eclairage system based on luminance and chromaticity coordinates)

By setting the color model, your application can specify color values relative to the desired color model. Since most workstations use only one color model (usually the RGB color model), colors are automatically converted from the values of the color model, specified in the **GPCML** call, to the workstation's real color model.

When converting `CIELUV` color components to another color model (typically RGB), the `CIELUV` values are clipped at a constant luminence to create valid `RGB` values. If your application specifies `CIELUV` values that are clipped, inquiring the color value will return the `CIELUV` values of the `RGB` color created by the conversion clipping.

The second way to define colors is to bypass the rendering color table by specifying colors directly as color vectors. *Direct color* vectors are interpreted according to the current Direct Color Model and can be set using the Set Direct Color Model **(GPDCM)** subroutine. The direct color model can be `RGB`, `HSV`, `CMY`, or `CIELUV`, and only effects direct color setting structure elements such as Set Polyline Color Direct.

The color definition phase is summarized in the following figure. The result of this phase is a color vector which is passed to the rendering pipeline.

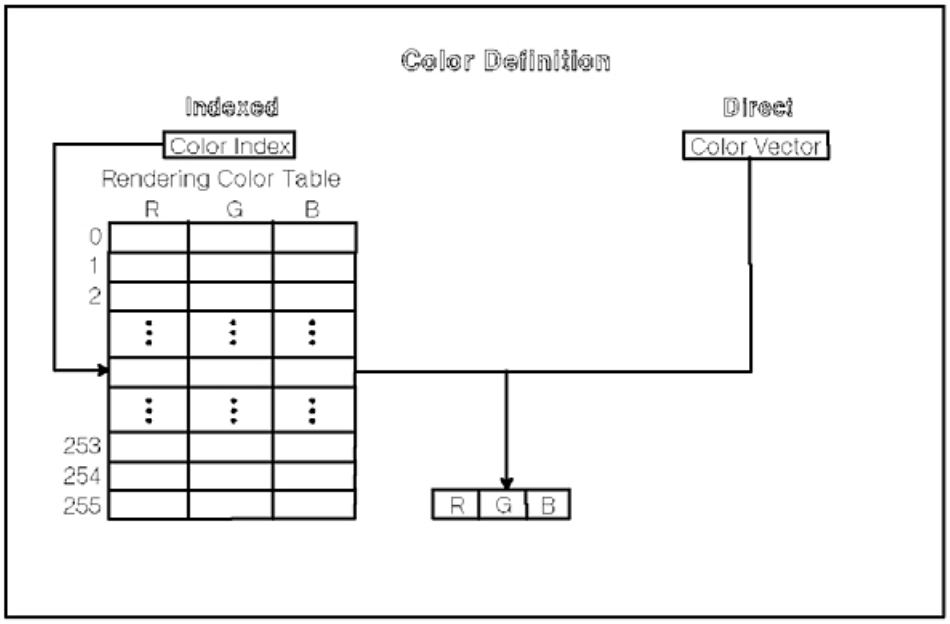


Figure 86. Color Definition Phase. This diagram shows how a color vector is derived from the rendering color table. A color index is supplied that references into the rendering color table. In this diagram, each entry in the rendering color table contains RGB values. The color index corresponds to one of these entries. Once the entry is selected by the color index, the RGB values are determined. These RGB values compose the color vector which is passed to the rendering pipeline.

Rendering

The input to the rendering pipeline is a color vector. This vector is processed by the rendering pipeline as described in Chapter 16. Rendering Pipeline. The rendering pipeline processes color according to the rendering color model in the current color processing table entry. The Rendering Color Model can be `1=RGB_NORMAL` or `2=RGB_B_ONLY`. In the `1=RGB_NORMAL` model, all three components of the color are manipulated by the rendering pipeline. In the `2=RGB_B_ONLY` model, only the blue component is manipulated by the rendering pipeline. The red and green components pass through the rendering pipeline unchanged. An example of how `2=RGB_B_ONLY` model is used is presented in Color Processing Examples.

A color processing table entry can be selected using the Set Color Processing Index **(GPCPI)** subroutine. A color processing table entry contains the color processing model (`1=RGB_NORMAL` or `2=RGB_B_ONLY`), quantization method, and quantization parameters. Color quantization is discussed in the next section.

All workstations have a color processing table with at least one entry, entry zero, which contains the default color processing information and cannot be changed. To set a color processing table entry your application can call the Set Color Processing Representation **(GPCPR)** subroutine. To determine the color processing capabilities of a workstation, your application should use the Inquire Color Processing Facilities **(GPQCPF)** subroutine.

See the following figure:

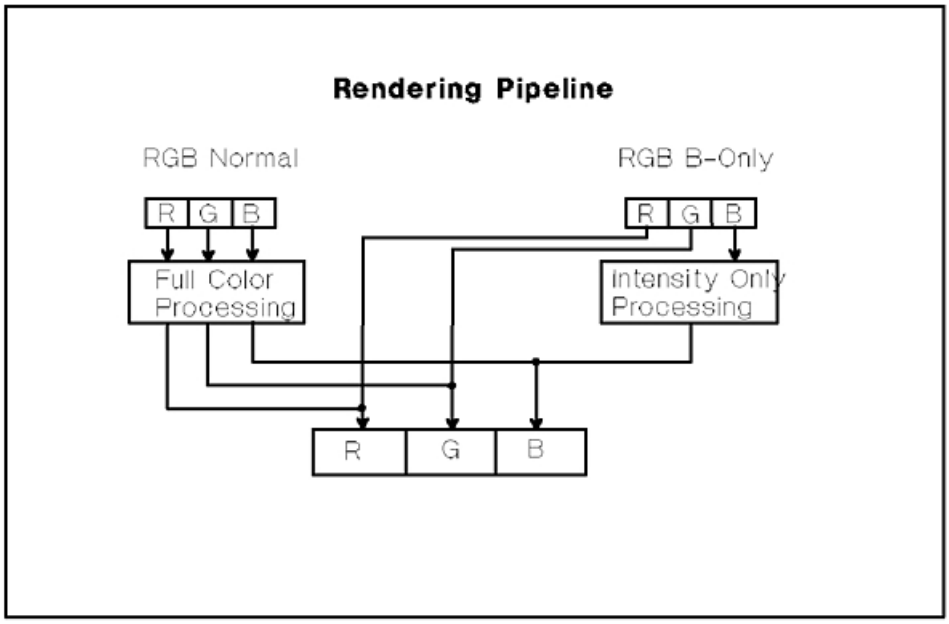


Figure 87. Rendering Color Models. This diagram shows the two possible rendering color models that can be in the color processing table entry. The RGB Normal color model performs color processing on each of the color components (R, G, and B) before going through the rendering pipeline. The RGB B-Only color model only processes the intensity of the B component. The R and G components are passed to the rendering pipeline without any change.

Color Quantization

Color quantization is defined as the process or method of mapping a specified color to a displayable color on the target workstation. For the graPHIGS API, the Quantization Method currently defined is `BITWISE` quantization.

Note: Some workstations have limited color resolution. That is, they are limited in the number of colors they can display at any one time. Shaded images, for example, can appear faceted on a workstation with limited resolution. An alternative is the use of a technique called *dithering* to improve the image output quality. For some workstations with limited color resolution, dithering is always on.

`BITWISE` quantization involves the following set of color quantization parameters:

N_r the number of bits of red to be used in the final color value

N_g the number of bits of green to be used in the final color value

N_b the number of bits of blue to be used in the final color value

N_p is an illegal DCF control word for a 32-bit string which defines the padding bits to be used in the final color value.

The use of these parameters depends on the type of frame buffer used by the workstation. There are two types of frame buffers: component and indexed. A *component frame buffer* consists of three conceptually separate color component buffers; one each for red, green, and blue as shown in the following figure. Each component of the buffer contains a color value for each pixel on the display surface. The color value represents an index into the workstation's *display* color table as shown in the following figure:

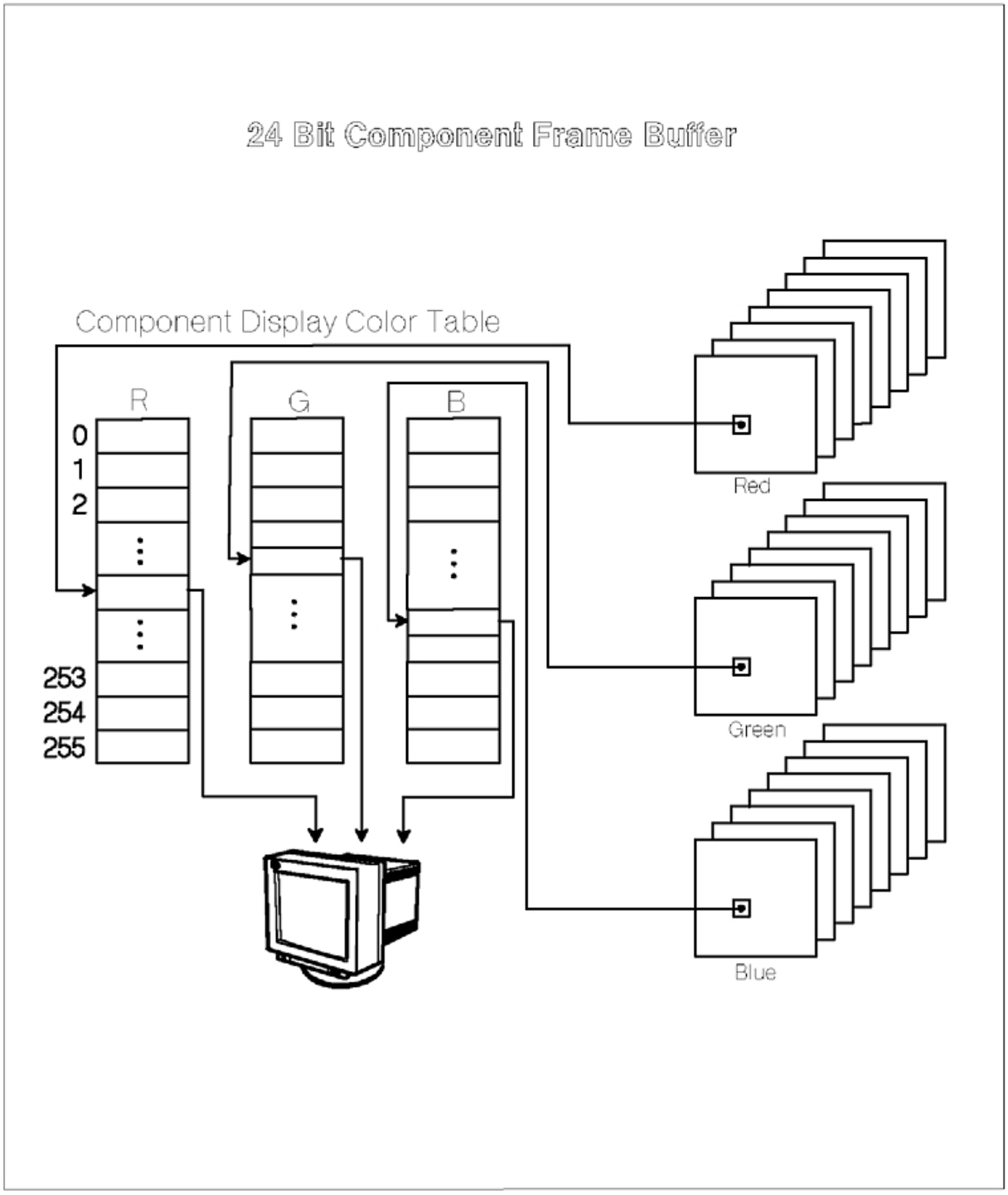


Figure 88. 24-Bit Component Frame Buffer. This diagram shows the configuration of a 24-Bit component frame buffer. There are three separate color component buffers. The component display color table contains 256 (0-255) color values for each component; that is, three tables, one for each component. This value can then be used by the display.

An **indexed frame buffer** contains only one display color table index for each pixel on the display surface as shown in the following figure, as opposed to the three color indexes provided by a component buffer.

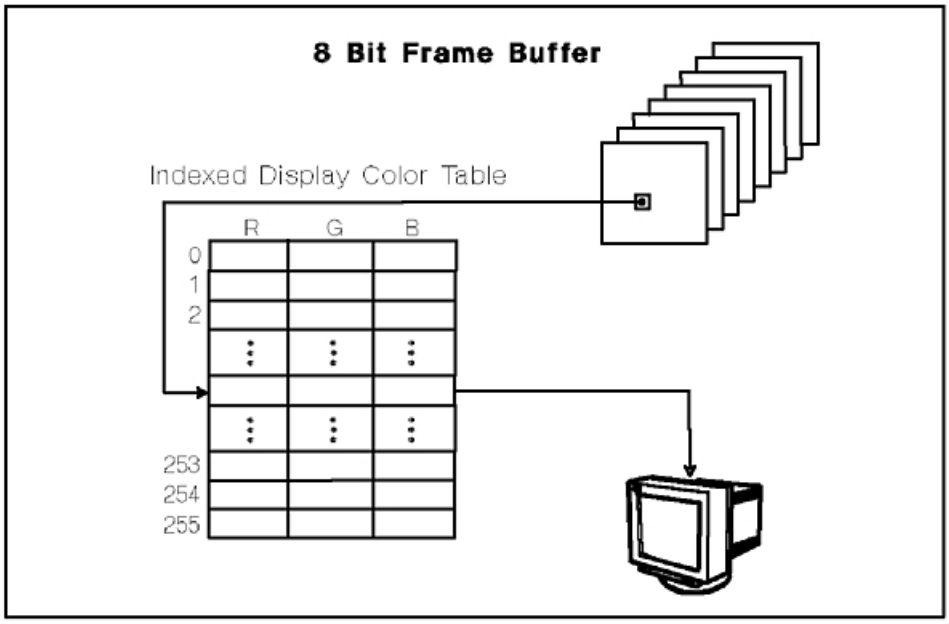


Figure 89. 8-Bit Indexed Frame Buffer. This diagram shows the configuration of an indexed frame buffer. In this case, there is only one index provided. The values provided in the display color table at this index are used by the display.

Component frame buffers provide millions more simultaneously displayable colors than do indexed frame buffers due to the separate indexing of the red, green, and blue components. For example, a workstation with a display color table of 256 entries and a component frame buffer with a bit depth of 24 bits (8 bits for each color component) will be capable of displaying from a palette of 16.7 million colors simultaneously. The same workstation would only be able to display 256 colors simultaneously using an index frame buffer.

Note that not all workstations support a display color table with 256 entries and some do not have user-definable display color tables. Therefore, to determine whether or not a workstation's display color table is modifiable, your application should call the Inquire Extended Color Facilities **(GPQXCF)** subroutine. For workstations which do have a modifiable display color table, your application can inquire the size of the table using the Inquire Color Table Characteristics **(GPQCCH)** subroutine. Your application can determine a workstation's frame buffer type and depth by calling the Inquire Frame Buffer Characteristics **(GPQFBC)** subroutine.

If the workstation has a component frame buffer, color quantization is performed on each red, green, and blue color component independently. Each color component is converted from a floating-point number between 0.0 and 1.0 to an N bit unsigned integer in the range 0 to $2^{N_x} - 1$ where N_x is the number of red, green, and blue bits (N_r, N_g, N_b) to be used in the final color value. The remainder of the bits are filled with the corresponding padding bits as shown in the following figure:

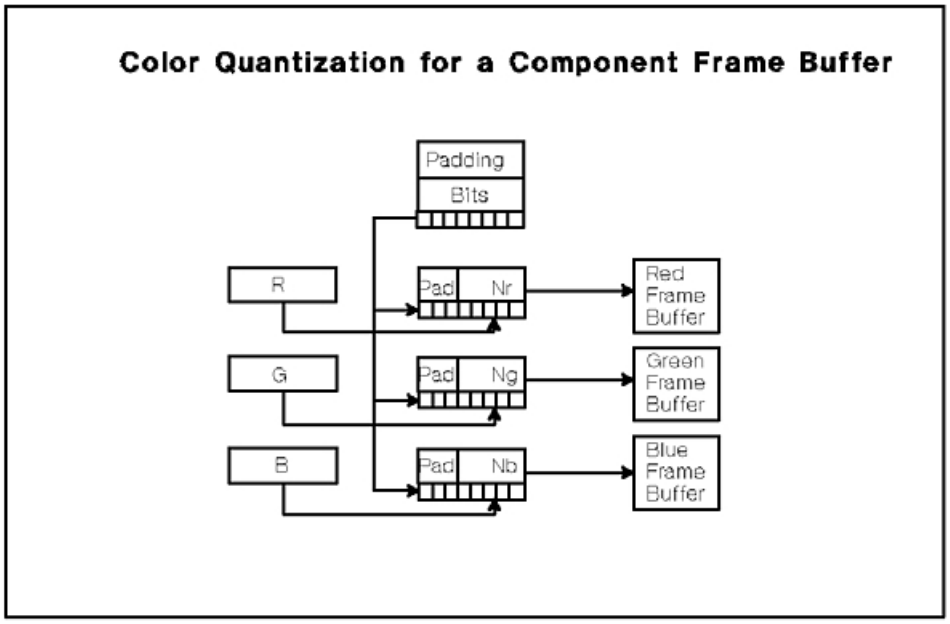


Figure 90. Color Quantization for a Component Frame Buffer. This diagram shows that padding bits are added to the front of each unsigned integer to be used in each final frame buffer component.

If the workstation has an indexed frame buffer, the `BITWISE` Quantization Method is used to reduce the `RGB` color vector down to a *N* bit integer where *N* is the bit depth of the frame buffer. Your application can determine the bit depth of an indexed frame buffer using the **GPQFBC** subroutine.

Color quantization for an indexed frame buffer is performed as follows:

- First, the red, green, and blue color components are converted from floating-point values between 0.0 and 1.0 to an *N* bit unsigned integer in the range 0 to $2^{N_x} - 1$ where N_x is the number of red, green, and blue bits (N_r, N_g, N_b) to be used in the final color value.
- Next, an *N* bit unsigned integer is formed using N_b bits of blue data, N_g bits of green data, and N_r bits of red data as shown in the following figure. Any bits that are left over are filled with the corresponding high-order bits from the padding bit data.

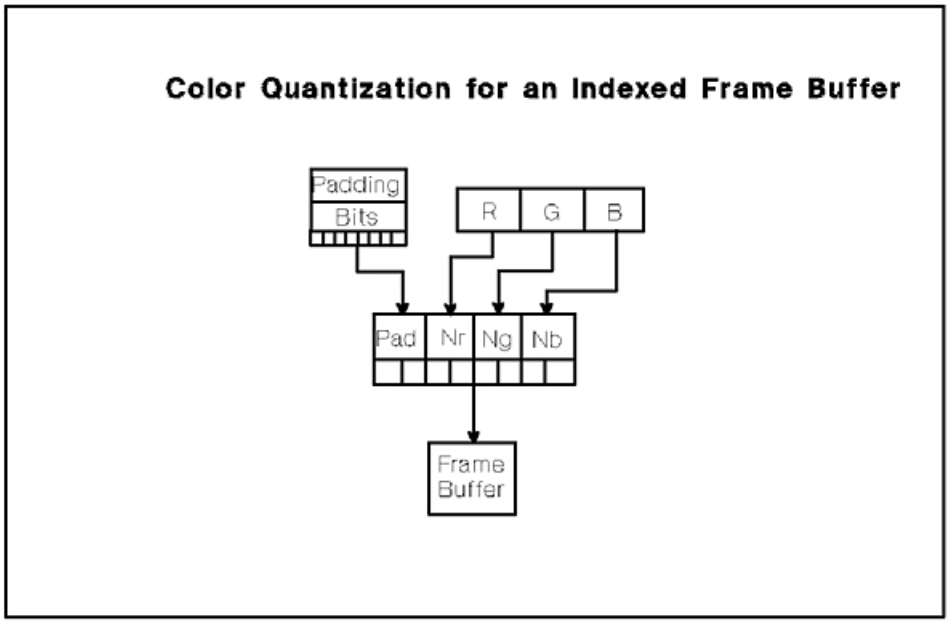


Figure 91. Color Quantization for an Indexed Frame Buffer. This diagram shows how padding bits are added to the front of the formed unsigned integer to create the value for the frame buffer.

The *N* bits of data are then written to the frame buffer as an integer index into the workstation's display color table.

Note: On some workstations, the ordering of the N_r and N_b bits within the quantized color value is reversed, resulting in `BGR` color quantization instead of `RGB` color quantization. The reversing of the red and blue bits affects how colors are quantized, including colors taken from the rendering color table. Initializing the default color table with a BGR color ramp illustrates how to set up a `BGR` rendering color table ramp for the 8-bit 3D Color Graphics Processor available for the RS/6000. Your application can determine whether a workstation performs `BGR` or `RGB` quantization through the Inquire List of Color Facilities **(GPQLCF)** subroutine.

Default Color Processing Configurations

To maintain upward compatibility with Version 1 of the graPHIGS API, all workstations by default process colors in a way similar to an IBM 5080. This means that to perform lighting, shading, depth cueing and direct color processing correctly, a workstation's color tables and color processing modes will have to be modified. Modifying the way a workstation processes colors is discussed in Color Processing Examples. The following sections discuss how individual workstation configurations process color by default.

To improve performance and speed in processing direct colors, a workstation PROCOPT can be set that causes a workstation to initialize the display color table differently. If the DIRCOLOR PROCOPT is specified, the workstation will initialize the color table components using values that range uniformly from the minimum component value to the maximum component value (linear ramps of the color components).

GDDM Devices

All of the various GDDM devices have a predefined display color table with either 8 or 16 entries. Because the entries in the display color table are fixed, your application can only change the contents of the rendering color table using either the Set Color Representation **(GPCR)** or the Set Extended Color Representation **(GPXCR)** subroutines. The colors entered into the rendering color table and direct colors specified by your application are mapped automatically to the closest approximation in the display color

table using a 1=WORKSTATION_DEPENDENT Quantization Method. This means that the use of direct color will work without having to change the default configuration.

5080 Workstations

All of the various 5080 models have an indexed frame buffer that is up to 8 bits deep. One of the bits is reserved by the graPHIGS API for input device echoes and prompts, leaving up to 7 available bits for use by your application. The actual number of available bits can be inquired using the Inquire Frame Buffer Characteristics **(GPQFBC)** subroutine.

By default, a 5080 workstation with 7 available frame buffer bit planes has a display color table with 128 entries and a rendering color table with 256 entries. The default Color Quantization Method is BITWISE 2-3-2-0 which means that 2 bits of red, 3 of green, 2 of blue, and no padding bits are used to quantize color vectors into 7 bit color indexes.

By default, the display color table is loaded as follows:

Index	***Color (R,G,B)***
0	Black (0.0, 0.0, 0.0)
1	White (1.0, 1.0, 1.0)
2	Red (1.0, 0.0, 0.0)
3	Green (0.0, 1.0, 0.0)
4	Blue (0.0, 0.0, 1.0)
5	Yellow (1.0, 1.0, 0.0)
6	Magenta (1.0, 0.0, 1.0)
7	Cyan (0.0, 1.0, 1.0)
8-127	White (1.0, 1.0, 1.0)

The rendering color table is loaded to produce a one-to-one mapping between the rendering color table and display color table. For example, entry 5 in the rendering color table contains the color vector (0.000, 0.143, 0.333) which when quantized using 2-3-2-0 quantization will produce the integer index 5. So, a 2-3-2-0 color table in combination with 2-3-2-0 color quantization produces the indexes 0 thru 127 for the rendering color table entries 0 thru 127. By default, entries 128 through 255 in the rendering color table map to index 1 in the display color table.

The default Rendering Color Model for a 5080 workstation is 1=RGB_NORMAL.

6090 without the EPM Feature

The Base 6090 without the Extended Pixel Memory (EPM) feature, has an indexed frame buffer that is 8 bits deep and therefore has a display color table with 256 entries. The default Color Quantization Method is BITWISE 3-3-2-0. 3-3-2-0 quantization means that 3 bits of red, 3 of green, 2 of blue, and no padding bits are used to quantize color vectors into 8-bit color indexes.

By default the display color table is loaded as follows:

Index	***Color (R,G,B)***
0	Black (0.0, 0.0, 0.0)
1	White (1.0, 1.0, 1.0)
2	Red (1.0, 0.0, 0.0)
3	Green (0.0, 1.0, 0.0)
4	Blue (0.0, 0.0, 1.0)
5	Yellow (1.0, 1.0, 0.0)
6	Magenta (1.0, 0.0, 1.0)
7	Cyan (0.0, 1.0, 1.0)
8-255	White (1.0, 1.0, 1.0)

The default color processing model is RGB_NORMAL and the input color table by default is loaded such that each input color table entry maps to the corresponding output color table entry. This is accomplished by loading entries 0 thru 255 with a 3-3-2-0 color table. A 3-3-2-0 color table in combination with 3-3-2-0 color quantization produces the indexes 0 thru 255 for rendering color table entries 0 thru 255.

6090 with the EPM Feature

When equipped with the Extended Pixel Memory (EPM) feature, the 6090 has a component frame buffer that is 24 bits deep. Like the base 6090, a 6090 with EPM has a display color table with 256 entries but the default Color Quantization Method set to BITWISE 8-8-8-0. 8-8-8-0 quantization means that color values are quantized such that each component of the frame buffer will receive an index into the display color table. By default, the display color table is loaded with the same values as a base 6090.

As with the base 6090, the default color processing model is RGB_NORMAL and the input color table by default is loaded such that each input color table entry maps to the corresponding output color table entry.

Color Processing Examples

In some cases you may wish to modify the way a workstation processes colors. For example, you may wish to define a range of colors for shading objects and another range to be used for the display of menus and other screen colors. In general, the default color processing configuration of a workstation will not be sufficient to handle shading and the use of direct color. The remainder of this chapter addresses a few of the typical color processing scenarios you may face when developing a graPHIGS API application. The scenarios are organized based on the following hardware configurations the graPHIGS API currently supports:

- GDDM devices with up to 16 available colors
- 5080 with an indexed frame buffer
- 6090 with an indexed frame buffer
- 6090 with a component frame buffer
- 8-bit 3D Color Graphics Processor

GDDM Devices

As discussed in the previous section, a GDDM device, such as a 3279 terminal, is limited to a maximum of 16 predefined colors in the display color table. To provide direct color support on a GDDM type device, the graPHIGS API maps color vectors to the closest available color using a 1=WORKSTATION_DEPENDENT Color Quantization Method. No other quantization methods are supported. The color vectors are taken either from the rendering color table when color indexes are used or directly when direct color vectors are supplied by your application. Therefore, your application does not need to perform any special processing in order to use the direct color support provided by the graPHIGS API on GDDM type devices.

5080 Workstations

The various models of the 5080 support only indexed frame buffers of up to 8 bits per pixel. The actual number of bits can be inquired using the Inquire Frame Buffer Characteristics **(GPQFBC)** subroutine. The graPHIGS API reserves one of the bit planes for input device echoing and prompts leaving up to 7 bits for your use. To use direct color on a 5080 your application must redefine the 5080's display color table to match the default color quantization method used for the workstation. For example, the default color quantization method used for a 5080 with 7 available bit planes is 2-3-2-0 quantization. 2-3-2-0 quantization means that 2 bits of red, 3 of green, 2 of blue, and no padding bits are used to quantize a direct color vector into a 7 bit display color table index. The FORTRAN subroutines shown in the figures, Initializing a display color table for direct color processing, Calculating color table entries for an index frame buffer, and Calculating color table entries for a component frame buffer, show how a display color table can be initialized to make use of direct color. The subroutines perform the necessary inquires to determine the type and size of the workstation's frame buffer in order to initialize the display color table correctly.

Figure: Initializing a display color table for direct color processing.

Example Subroutine

```
*********************************************************************
*                            S E T C T
*********************************************************************
*   FUNCTION:  INITIALIZES A WORKSTATION'S DISPLAY COLOR TABLE
*              BASED ON THE NUMBER OF AVAILABLE BIT PLANES IN
*              THE WORKSTATION'S FRAME BUFFER.
*********************************************************************
*  CALLED BY:  APPLICATION PROGRAM
*********************************************************************
* PARAMETERS:  (INPUT)
*   INTEGER WSID - WORKSTATION IDENTIFIER
*********************************************************************
* PARAMETERS:  (OUTPUT)
*   NONE
*********************************************************************



      SUBROUTINE SETCT(WSID)

      INTEGER WSID
      REAL    COLOR(3, 256)
      INTEGER ERRIND, ORG, N, DEPTH(3), CIDLEN
      CHARACTER*8 CONNID, WSTYPE
      INTEGER CTID, START, CTSIZE
      PARAMETER (CTID = -1, START = 0)

C     INQUIRE REALIZE CONNECTION AND TYPE
      CALL GPQRCT(WSID, LEN(CONNID), ERRIND, CIDLEN, CONNID, WSTYPE)

      IF (ERRIND .EQ. 0) THEN
C       INQUIRE FRAME BUFFER CHARACTERISTICS
        CALL GPQFBC(WSTYPE, ERRIND, ORG, N, DEPTH)
        IF (ERRIND .EQ. 0) THEN
          IF (N .EQ. 1) THEN
C           CALCULATE COLOR TABLE VALUES FOR INDEXED FRAME BUFFER
            CALL GETCTI(DEPTH, CTSIZE, COLOR)
          ELSE IF (N .EQ. 3) THEN
C           CALCULATE COLOR TABLE VALUES FOR COMPONENT FRAME BUFFER
            CALL GETCTC(DEPTH, CTSIZE, COLOR)
          ELSE
C           SOMETHING IS WRONG
            WRITE(*,*)'NUMBER OF FRAME BUFFER COMPONENTS NOT 1 OR 3'
            RETURN
          ENDIF
        ELSE
C         GPQFBC FAILED SO ISSUE ERROR
          WRITE(*,*)'ERROR RETURNED BY GPQFBC (ERRIND = ',ERRIND,')'
          RETURN
        ENDIF
      ELSE
C       GPQRCT FAILED SO ISSUE ERROR
        WRITE(*,*)'ERROR RETURNED BY GPQRCT (ERRIND = ',ERRIND,')'
        RETURN
      ENDIF

C     SET COLOR TABLE
      CALL GPXCR(WSID, CTID, START, CTSIZE, COLOR)

      RETURN
      END
```

Figure: Calculating color table entries for an index frame buffer.

Example Subroutine

```
************************************************************************
*                            G E T C T I                               *
************************************************************************
*   FUNCTION:  CALCULATES COLOR TABLE VALUES BASED ON THE NUMBER OF
*              AVAILABLE BIT PLANES IN A WORKSTATION'S INDEXED FRAME
*              BUFFER.
************************************************************************
*  CALLED BY:  SETCT - SET COLOR TABLE
************************************************************************
* PARAMETERS:  (INPUT)
*   INTEGER DEPTH      - BIT DEPTH OF INDEXED FRAME BUFFER
************************************************************************
* PARAMETERS:  (OUTPUT)
*   INTEGER CTSIZE     - SIZE OF THE COLOR TABLE
*   REAL    COLOR(3,*) - COLOR TABLE VALUES
************************************************************************
```

```
      SUBROUTINE GETCTI(DEPTH, CTSIZE, COLOR)

      INTEGER DEPTH, CTSIZE
      REAL    COLOR(3,*)
      INTEGER BITS(3,8)
      INTEGER R, G, B, REND, GEND, BEND

C     QUANTIZATION DATA
      DATA BITS / 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 2, 1,
     >            2, 2, 1, 2, 2, 2, 2, 3, 2, 3, 3, 2 /

C     CALCULATE NUMBER OF LOOPS FOR EACH COLOR COMPONENT
      REND = 2**BITS(1,DEPTH) - 1
      GEND = 2**BITS(2,DEPTH) - 1
      BEND = 2**BITS(3,DEPTH) - 1

C     INITIALIZE COLOR TABLE INDEX
      I = 0

C     LOOP TO CALCULATE COLOR VECTORS
      DO 300 R = 0, REND
        DO 200 G = 0, GEND
          DO 100 B = 0, BEND
            I = I + 1
            IF (REND.EQ.0) THEN
              COLOR (1,I) = 0.0
            ELSE
              COLOR (1,I) = FLOAT(R) / FLOAT(REND)
            ENDIF
            IF (GEND.EQ.0) THEN
              COLOR (2,I) = 0.0
            ELSE
              COLOR (2,I) = FLOAT(G) / FLOAT(GEND)
            ENDIF
            IF (BEND.EQ.0) THEN
              COLOR (3,I) = 0.0
            ELSE
              COLOR (3,I) = FLOAT(B) / FLOAT (BEND)
            ENDIF
 100      CONTINUE
 200    CONTINUE
 300  CONTINUE

C     RETURN SIZE OF COLOR TABLE
      CTSIZE = I

      RETURN
      END
```

Figure: Calculating color table entries for a component frame buffer.

Example Subroutine

```
*********************************************************************
*                           G E T C T C
*********************************************************************
*   FUNCTION:  CALCULATES COLOR TABLE VALUES BASED ON THE NUMBER OF
*              AVAILABLE BIT PLANES IN A WORKSTATION'S COMPONENT FRAME
*              BUFFER.
*********************************************************************
*  CALLED BY:  SETCT - SET COLOR TABLE
*********************************************************************
* PARAMETERS:  (INPUT)
*   INTEGER DEPTH(3) - FRAME BUFFER BIT DEPTHS
*********************************************************************
* PARAMETERS:  (OUTPUT)
*   INTEGER CTSIZE     - SIZE OF THE COLOR TABLE
*   REAL    COLOR(3,*) - COLOR TABLE VALUES
*********************************************************************



      SUBROUTINE GETCTC(DEPTH, CTSIZE, COLOR)

      INTEGER DEPTH(3), CTSIZE
      REAL    COLOR(3,*)
      REAL    VALUE

C     INITIALIZE COLOR TABLE SIZE
      CTSIZE = 2xxDEPTH(1)

C     LOOP TO CALCULATE COLOR TABLE VALUES
      DO 100 I = 1, CTSIZE
C       CALCULATE ITH COLOR VALUE
        VALUE = FLOAT(I-1) / FLOAT(CTSIZE - 1)
C       COPY ITH COLOR VALUE TO COLOR ARRAY
        COLOR(1,I) = VALUE
        COLOR(2,I) = VALUE
        COLOR(3,I) = VALUE
 100  CONTINUE

      RETURN
      END
```

6090 with the EPM Feature

The 6090 with the Extended Pixel Memory (EPM) feature is very easy to use for true color shading, lighting, depth cueing, and direct color. With true color, your application can display any color by choosing from 256 shades of red, 256 shades of green, and 256 shades of blue. This is still an approximation, but the error is not detectable by the human eye. To use true color your application must first, load the display color table with a ramp from 0.0 to 1.0 as shown in the figure, Initializing a display color table with a color ramp.

Next, your application should load the rendering color table with any indexed colors that the application needs. The 6090 will then be ready for your application to use true color.

6090 without the EPM Feature

A 6090 equipped with a shading card but not the EPM feature is still capable of lighting, shading, and depth cueing. Without the shading card, the only advanced color functionally available to your application is direct color in which case your application can treat the 6090 as a 5080 with an 8-bit frame buffer. To setup a 6090 without the EPM or shading card features read the section titled 5080 Workstations.

If the 6090 available to your application does have a shading card but not the EPM feature, your application can produce shaded images, but not true color images because the total number of colors on

the screen at one time is limited to 256. Therefore, you must choose between the number of colors and number of shades of each color your application will use. That means the application can have a white shaded image with 256 shades of grey, or a two color shaded image with 128 shades of each color, etc.. As the number of colors goes up, the accuracy of shading goes down, which results in *Mach Banding* Mach bands are noticeable bands of constant color caused by the low number of colors available to shade the object. The number of colors available for shading is also reduced by the need for indexed colors for menus, views, etc. The following scenarios present some of the ways your application can maximize the use of the limited number of simultaneous colors.

Scenario 1 - One Color

For the first scenario assume your application is designed to display a shaded image in one base color using all 256 color table entries. To do this your application can define an entry in the color processing table using the Set Color Processing Representation **(GPCPR)** subroutine. The entry to be defined will specify bitwise color quantization using 8 bits of red, 0 bits of both green and blue and no padding bits (8-0-0-0 quantization). Note that 8-0-0-0 quantization simply means that the red component of a color is in effect the only component that is quantized into an index. The index is then used to select the actual color to be displayed from the display color table. So, the display color table must be loaded with a ramp from black (0.0, 0.0, 0.0) at entry 0 to the desired color at entry 255. If the desired color is R_i, G_i, B_i, then the subroutine shown in the figure, Initializing a display color table with a color ramp, will initialize the display color table correctly.

Figure: Initializing a display color table with a color ramp.

Example Subroutine

```
*********************************************************************
*                            C L R R M P
*********************************************************************
*   FUNCTION:  INITIALIZES A WORKSTATION'S DISPLAY COLOR TABLE WITH
*              A RAMP FROM (0.0, 0.0, 0.0) TO (R, G, B).  THE
*              WORKSTATION IS ASSUMED TO HAVE A COLOR TABLE WITH 256
*              ENTRIES.
*********************************************************************
*  CALLED BY:  APPLICATION PROGRAM
*********************************************************************
* PARAMETERS:  (INPUT)
*   INTEGER WSID   - WORKSTATION IDENTIFIER
*   REAL R, G, B   - RED, GREEN, AND BLUE COMPONENTS OF THE TARGET
*                    COLOR
*********************************************************************
* PARAMETERS:  (OUTPUT)
*   NONE
*********************************************************************
```

```
      SUBROUTINE CLRRMP(WSID, R, G, B)

      INTEGER WSID
      REAL    R, G, B
      INTEGER CTID, CTSIZE, START
      PARAMETER (CTID = -1, CTSIZE = 256, START = 0)
      REAL    COLOR(3, CTSIZE)

      DO 100 I = 1, CTSIZE
        COLOR(1,I) = R * (I - 1) / FLOAT(CTSIZE - 1)
        COLOR(2,I) = G * (I - 1) / FLOAT(CTSIZE - 1)
        COLOR(3,I) = B * (I - 1) / FLOAT(CTSIZE - 1)
100   CONTINUE

      CALL GPXCR(WSID, CTID, START, CTSIZE, COLOR)

      RETURN
      END
```

Next, your application should call the Set Color Processing Index **(GPCPI)** subroutine to insert a Color Processing Index attribute into the structure to be rendered. The index should select the color processing table entry defined using the **GPCPR** subroutine. Your application must also define the light, surface, and depth cue colors as white (1.0, 1.0, 1.0).

If you need the use of indexed colors, your application can load the output color table with a color ramp that goes from entry 20 thru 255 instead of 0 thru 255. This gives your application 20 indexed colors (entries 0 through 19) that can be used for menus and other screen colors. To prevent these twenty colors from showing up at the darker parts of the shaded image, give the shaded surface an ambient reflection coefficient greater than 0.1 using the Set Surface Properties **(GPSPR)** and Set Back Surface Properties **(GPBSPR)** subroutines. Also make sure there is an ambient light defined with a color of white using the Set Light Source Representation **(GPLSR)** subroutine. This means that the darkest part of the surface will produce an intensity that maps to entry 25 in the display color table.

The structures that make use of the indexed colors must also contain a color Processing Index of 0 to get default color processing. This assumes that the rendering color table is still loaded with default 3-3-2-0 color table values.

Note: In this case, indexed color will only work for primitives that have not had lighting, shading, or depth cueing applied to them.

Scenario 2 - Two Colors

To use two colors your application can make use of the `RGB_B_ONLY` color processing method with 1-0-7-0 color quantization. With `RGB_B_ONLY`, only the blue component of a color is processed by the rendering pipeline. The red and green components pass through the pipeline untouched. By using 1-0-7-0 quantization, the blue component will control the intensity of a color and the red component will select one of the two colors. The green component of the color vector is ignored. For example, suppose your application defines the color of a polyline to be (0.0, 0.5, 0.25). After depth queueing using `RGB_B_ONLY` processing one of the pixels defined by the polyline might have a color of (0.0, 0.5, 0.2). Note that only the blue component of the color changed. Using 1-0-7-0 quantization, the color vector (0.0, 0.5, 0.2) will be mapped to the integer 26 which represents an index in the display color table. If the red component had been set to 1.0 instead of 0.0 the integer index would have been 154 instead of 26, a difference of 128 or half of the size of the display color table. So, the red component can be used to switch between the two halves of the display color table. If a primitive's color attribute is defined with the red component equal to 1.0 then the upper half of the display color table will be used to render the primitive. If the red component is 0.0 then the lower half of the display color table will be used. By defining the two halves of the display color table with two color ramps, one from 0 thru 127 and one from 128 thru 255, primitives can be shaded with two different base colors.

To control which color ramp will be used to render a surface, your application can set the interior color of the surface with one of two color vectors. To get the first ramp, your application should specify a direct interior color of (0.0, 1.0, 1.0), and to get the second ramp, your application should specify a direct color of (1.0, 1.0, 1.0). All other colors associated with rendering should remain white as in scenario 1.

To mix indexed colors simultaneously with shading, your application should use the same method outlined in scenario 1 except the indexed colors will be in the range 0 through 9 and 128 through 137 instead of 0 through 19.

Scenario 3 - Four Colors

The four color scenario is similar to the two color scenario except that 1-1-6-0 color quantization should be used instead of 1-0-7-0 quantization. Your application can then define four color ramps: 0 through 63, 64 through 127, 128 through 191, and 192 through 255 in the display color table of the workstation.

To control which color ramp will be used for shaded surfaces, use the following direct interior colors: 0.0 0.0 1.0 for ramp 1, 0.0 1.0 1.0 for ramp 2, 1.0 0.0 1.0 for ramp 3, and 1.0 1.0 1.0 for ramp 4. Again, the method for using both indexed colors and shading simultaneously as outlined in scenario 1 will allow your application to use the following color indexes for indexed colors: 0 through 4, 64 through 68, 128 through 132, and 192 through 196.

8-Bit 3D Color Graphics Processor

A RS/6000 3D Color Graphics Processor with dual 8-bit frame buffers is capable of lighting, shading, and depth cueing. The quantization phase assumes the color components are ordered blue, green, red (BGR). One way to initialize the input color table is with a BGR BITWISE 3-3-2-0 color ramp as shown in the figure, Initializing the default color table with a BGR color ramp.

Table 4. Initializing the default color table with a BGR color ramp.

Example Subroutine

```
*********************************************************************
*                            B G R R M P
*********************************************************************
*   FUNCTION:  INITIALIZES A WORKSTATION'S DISPLAY COLOR TABLE WITH
*              A RAMP FROM (0.0, 0.0, 0.0) TO (B, G, R).  THE
*              WORKSTATION IS ASSUMED TO HAVE A COLOR TABLE WITH 256
*              ENTRIES WITH PIXEL COLOR COMPONENTS ORDERED BGR
*********************************************************************
*  CALLED BY:  APPLICATION PROGRAM
*********************************************************************
* PARAMETERS:  (INPUT)
*   INTEGER WSID   - WORKSTATION IDENTIFIER
*   REAL B, G, R   - BLUE, GREEN, AND RED COMPONENTS OF THE TARGET
*                    COLOR
*********************************************************************
* PARAMETERS:  (OUTPUT)
*   NONE
*********************************************************************
```

Table 4. Initializing the default color table with a BGR color ramp. (continued)

```
      SUBROUTINE BGRRMP(WSID, B, G, R)

      INTEGER WSID
      REAL    B, G, R
      INTEGER INDEX, I, J, K
      INTEGER START, CTSIZE
      PARAMETER (START = 0, CTSIZE = 256)
      REAL    COLOR(3, CTSIZE)

      INDEX = 1

      DO 300 I = 0, 3
        DO 200 J = 0, 7
          DO 100 K = 0, 7
            COLOR(1,INDEX) = B * FLOAT(K) / 7.0
            COLOR(2,INDEX) = G * FLOAT(J) / 7.0
            COLOR(3,INDEX) = R * FLOAT(I) / 3.0
            INDEX = INDEX + 1
100       CONTINUE
200     CONTINUE
300   CONTINUE

      CALL GPCR(WSID, START, CTSIZE, COLOR)

      RETURN
      END
```

Frame Buffer Manipulation

Many workstations use frame buffers to perform the rendering of a picture which is to be displayed. The graPHIGS API has controls that allow the application to control the frame buffer to achieve certain effects during rendering. The following section describes controls available during the graPHIGS traversal and rendering process.

Your application has additional controls available when doing explicit traversal processing. These explicit controls allow your application to select frame buffers for display or for rendering processing. See Chapter 14. Explicit Traversal Control for more details of the frame buffer controls available when doing explicit traversal processing.

Frame Buffer Write Protect Mask

Some workstations are able to inhibit writing to selected frame buffer bit planes. This capability is useful when you need to control display priorities at the primitive level. By writing primitives with lower priorities and those with higher priorities into a lower and higher portion of the frame buffer respectively, and setting the display color table appropriately, your application can control display priorities of primitives independent of their traversal order.

To support this capability the Generalized Structure Element (GSE) Frame Buffer Protect Mask is supported on some workstations. A Frame Buffer Protect Mask structure element is created by the Set Frame Buffer Protect Mask **(GPFBM)** subroutine which takes a 32-bit bit string parameter specifying a frame buffer mask.

For an indexed frame buffer workstation, the least significant *M* bits are used for the actual mask where *M* is the bit depth of the frame buffer. For a component frame buffer, the least significant M_1 bits, the next least significant M_2 bits, and the next least significant M_3 bits are used for each component of the frame buffer, where M_1, M_2, and M_3 are the bit depths of the three frame buffer components.

When a frame buffer mask bit is set to 1, writing to the corresponding bit plane is inhibited. Therefore, the mask '00000000'X means that writing to all bit planes is permitted and the mask 'FFFFFFFF'X means that writing to all bit planes is inhibited.

The Frame Buffer Mask structure element is defined as a GSE and is not supported on all workstations. Even when the GSE is supported, the contents (mask bits) should be set in a `WORKSTATION_DEPENDENT` way because two workstations may have different frame buffer organizations. This means that if your application uses frame buffer masks it may not be portable between different workstations. Use of this capability should therefore be considered very carefully to prevent loss of application portability.

The initial frame protect buffer mask for a specific view can be set using the Set Extended View Representation **(GPXVR)** subroutine. The initial frame buffer protect mask is the traversal default for all images and structures displayed in the view. The mask does not affect the view shielding or border.

Direct Access to Frame Buffer

In the graPHIGS API, a frame buffer is accessed as a two-dimensional array of pixels, as shown in the following figure:

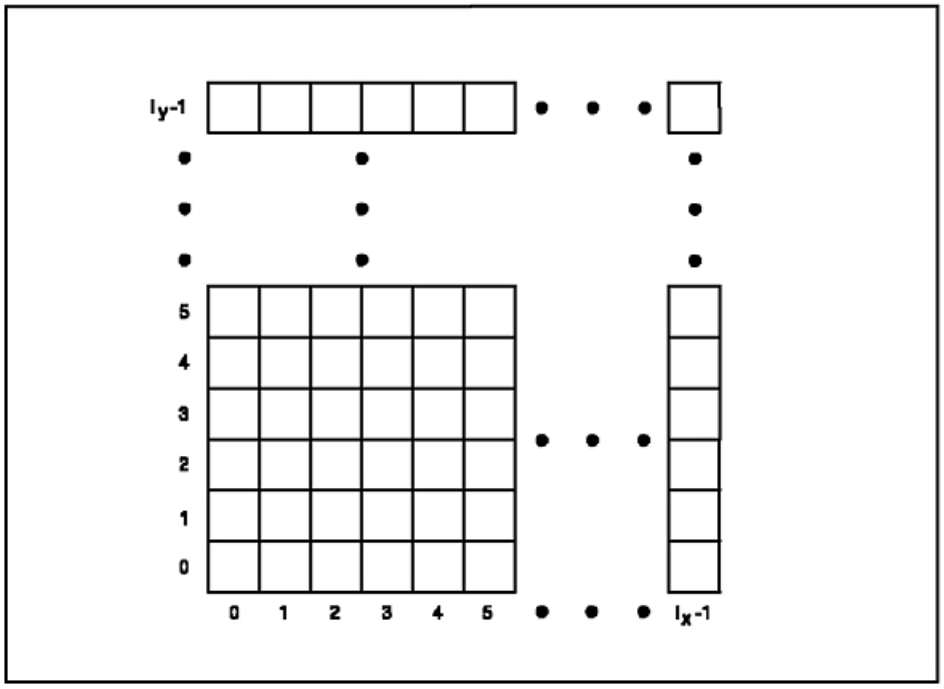


Figure 92. Accessing a frame buffer as an array of pixels. This diagram shows the configuration of a frame buffer accessed as a two-dimensional array of pixels. The array is of size I_X by I_y. The indices are from 0 to I_x - 1 and 0 to I_y - 1 respectively.

I_x and I_y are the Device Coordinates (DC) in raster units of the workstation. For every operation addressed to the frame buffer directly, the target area is defined as a rectangle by specifying its origin (position of the lower left pixel) and size. Let (O_x, O_y) and (S_x, S_y) be the origin and the size of the rectangle, respectively. The following relations must be satisfied:

$$0 <= O_x <= O_x + S_x - 1 <= I_x - 1$$
$$0 <= O_y <= O_y + S_y - 1 <= I_y - 1$$

A component frame buffer actually has multiple frame buffer components and each component must be accessed by a separate operation.

The only direct access that is currently provided allows your application to retrieve a frame buffer's contents. This is accomplished through the Read Frame Buffer **(GPRDFB)** subroutine. The parameters for this subroutine are:

- Workstation identifier.
- Frame buffer component number. This must be a valid frame buffer number which is in the range 1 to the number of frame buffer components available on the workstation. The number of frame buffer components can be determined by invoking the Inquire Frame Buffer Characteristics **(GPQFBC)** subroutine.
- Source rectangle origin (O_x, O_y) as described above.
- Rectangle size (S_x, S_y) as described above.
- Format of the application area in which the data is to be placed.
- Format dependent parameters.
- Target area in which the frame buffer data is to be placed.
- The location within the target area where the data is to be placed.

This subroutine call will return the content of the frame buffer from the target workstation after all updates currently in process have completed. This subroutine call will not initiate an update but will wait for a previously initiated one to complete. This means that the pixel data that is returned will correspond to the Display Status field in the workstation state list as follows:

1=CORRECT
The returned pixel data exactly corresponds to the graphical state of the workstation.

2=DEFERRED
The returned pixel data will be different from the graphical state of the workstation.

3=SIMULATED
The returned pixel data will contain a reasonable simulation of the graphical state of the workstation.

Note: This subroutine call does not initiate a new update process. It is your application's responsibility to serialize access to the workstation during the read frame buffer operation. This means that other application processes should not attempt to modify the workstation while the read frame buffer is being performed. To guarantee the state of the frame buffer, your application should call the Update Workstation **(GPUPWS)** subroutine, before calling the Read Frame Buffer subroutine.

Frame Buffer Comparison Options

You can use the Set Frame Buffer Comparison **(GPFBC)** subroutine to control the update of the frame buffer during traversal. Such controls allow your application to achieve certain special rendering effects such as *underpaint* and *line-on-line highlighting*.

The **GPFBC** subroutine allows specification of a mask value. This value selects certain bit planes of the frame buffer for testing. The contents of the selected bit planes are compared to a color value (also specified with the **GPFBC** subroutine). There are three comparison options.

1=NO_OPERATION
specifies no comparison.

2=WRITE_WHEN_EQUAL
replaces the pixel value currently in the frame buffer with the new pixel value being rendered. This operation performs an underpaint when the comparison value is the background color. New pixels replace background pixels, but do not replace the pixels of other primitives. Objects appear to be behind other objects, independent of their drawing priority and HLHSR processing.

3=WRITE_WHEN_NOT_EQUAL
replaces the pixel value in the frame buffer with the highlighting color value. Use this frame buffer comparison option to highlight the intersection of geometry on the screen.

The `WRITE_WHEN_NOT_EQUAL` option uses the color attribute you specified for processing in the Set Line-On-Line Color Direct **(GPLLCD)** or Set Line-On-Line Color Index **(GPLLCI)** subroutines. The resulting direct or index color overrides any color specified by the attribute color for the primitive or specified in its definition.

Chapter 18. Advanced Input and Event Handling

This chapter provides information to help you take advantage of the graPHIGS API's advanced input and event handling capabilities. It contains a discussion of logical and physical input devices, input device triggers, cursor shape control, and extensions to the PHIGS input model. A short review of the PHIGS input model will be presented followed by a discussion of the advanced graPHIGS API functions. If you are unfamiliar with using the graPHIGS API to obtain input, see Chapter 7. Input Devices located in the first part of this book.

The PHIGS Input Model

This section contains a short overview of the PHIGS input device model. Later portions of this chapter will discuss the graPHIGS API extensions to the model that provide you with greater control over input processing.

Logical Input Devices

In order to promote application portability, the PHIGS standard adopted the concept of *logical* input devices. The logical input device model shields your application programs from differences in hardware input capabilities by providing a consistent method of acquiring input.

Each logical input device has the concept of a *measure* The measure is a value determined by one or more physical input devices together with a *measure mapping* The measure reflects the current state of the logical input value and is continuously updated while the logical input device is active.

The PHIGS standard defines six logical input device classes. A class is defined by the measure returned by the device as described below:

Device
: Description

1=LOCATOR
: Usually implemented as a cursor controlled by a mouse, stylus, or puck, the measure of a locator device is a view identifier and the position of the screen cursor in World Coordinates (WC).

2=STROKE
: A stroke device generates a measure which consists of a view identifier and a set of points in World Coordinates (WC) defined by a screen cursor.

3=VALUATOR
: A valuator device provides scalar values within a specified range and is typically implemented with a dial.

4=CHOICE
: A choice device generates integer input based on a selection made from a button type device such as the function keys on a computer keyboard.

5=PICK
: A pick device is used to select graphical elements and, like the locator and stroke devices, is usually implemented with a screen cursor controlled by a mouse, stylus, or puck.

6=STRING
: A computer keyboard is typically used to implement a string device which supplies character data to an application program in the form of a character string.

Input Triggers

The logical input device model also includes the concept of triggers. A *trigger* is used to indicate a significant point in time when an operation is to be performed on the measure or the state of the logical input device. The logical input device model is illustrated in the following figure:

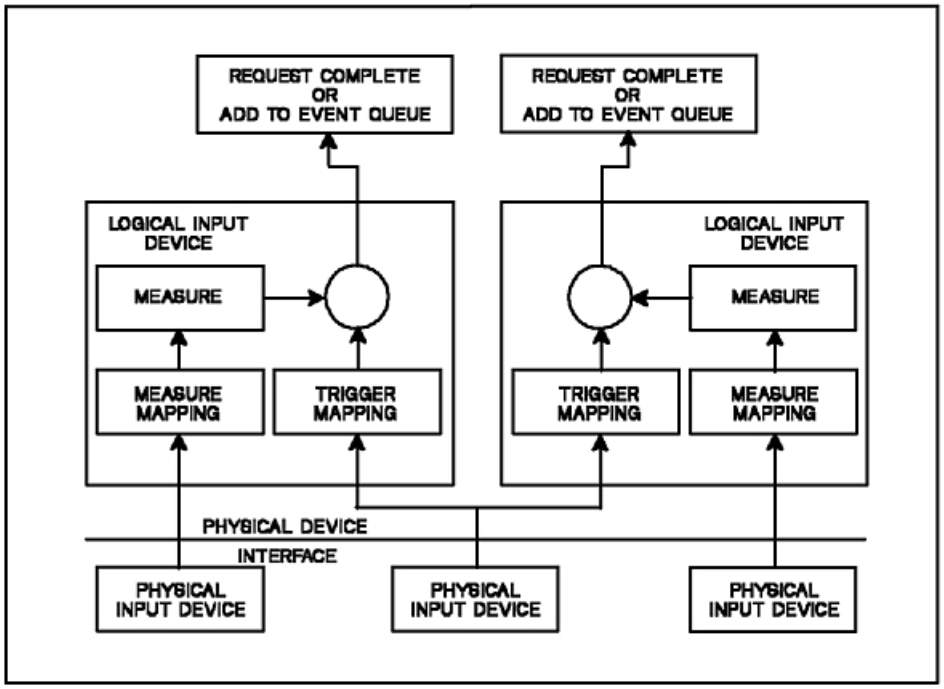


Figure 93. PHIGS Input Device Model. This diagram shows the three physical input devices and how they interface with the logical input device. Each physical device interfaces with the logical model and either measures mapping or filters through the trigger mapping. When this happens, the request is either completed or added to the event queue.

Input triggers are discussed in detail later in this chapter.

Input Echo

In addition to the measure and trigger, a logical input device may generate output to indicate the current state of the device's measure or to acknowledge the receipt of a trigger. This output is usually visible or audible and is called the input device `ECHO`.

Input Modes

As discussed in the first part of this book (Basic), the method of obtaining the measure of a device depends upon the device's Operating Mode. The three modes defined by the PHIGS standard are `1=REQUEST`, `2=SAMPLE`, and `3=EVENT` as defined in the following sections.

Request Mode Input

An input device in `REQUEST` mode is inactive until your application program specifically requests input from the device. Control will not return to your application program until the request is satisfied by the user triggering the device.

Sample Mode Input

When an input device is in `SAMPLE` mode the device is active meaning that it is ready to provide input. To obtain the input, your application program samples the device's current measure. The most common use of `SAMPLE` mode input is to determine the current measure of a `LOCATOR` or `VALUATOR` device. Other devices have limited utility in `SAMPLE` mode.

Event Mode Input

Like a device in SAMPLE mode, a device in EVENT mode is active but, instead of your application program sampling the device for input, the user triggers the device to generate the input. Because you can trigger the device at any time, the input is placed in an *event queue* to be processed when your application program is ready. EVENT mode input allows you to input data even if your application is busy doing something else.

To determine if an event has occurred your application calls the Await Event **(GPAWEV)** subroutine. If the event queue is empty, **GPAWEV** will wait for a specified number of seconds for an event to occur. If an event occurs during the wait or there is an event on the queue when **GPAWEV** is called, **GPAWEV** will return the following information:

Major code -
an integer defined by the class of the event

Event class -
an integer defining the type of event

Minor code -
an integer defined by the class of the event.

If the wait time elapses, the event class will be 0=NONE

In the PHIGS standard, only input devices can generate events. Therefore, the major code is defined as the workstation identifier, the event class is the input device class, and the minor code is the input device number. If the event class is not 0=NONE, your application can retrieve the event information from the current event report, using one of the graPHIGS API *get event* subroutine calls. For example, if the event class returned by **GPAWEV** indicates that the event was generated by a choice device (class = 4), your application would call the Get Choice **(GPGTCH)** subroutine to get the choice information from the event queue.

The concept of *events* has been extended in the graPHIGS API to define events that are not related to input devices as discussed in the following section.

Input Model Extensions

Event Extensions

In addition to input device events, the graPHIGS API can also generate and process events which are not related to input devices. This section describes each of those events, how they are generated, and how your application can use them. All events are categorized by the event class parameter of the Await Event **(GPAWEV)** subroutine as listed below:

<0=Not Used
Negative values are not used for the event class.

0=None
Zero is used to indicate that no events are in the event queue. When the event class is zero, the major code and minor code parameters are not set. This value may be returned in two cases, when a time out occurs, or when one or more asynchronous errors are reported by a nucleus to the graPHIGS API shell.

1-100=Input Device Events
This value range is used for normal input device events. When the event class is in this range, the major and minor codes contain a workstation identifier and a device number, respectively, and the current event report will contain an appropriate input device measure. The following classes are defined:

- 1=LOCATOR

- 2=STROKE
- 3=VALUATOR
- 4=CHOICE
- 5=PICK
- 6=STRING

101-200=Workstation Events

This value range is used for events related to the general status of a workstation. When the event class is in this range, the major code contains the workstation identifier of the workstation which generated the event. The minor code may or may not be set according to the individual event class. The following event classes are defined:

- 101: Link Switch Out

  This class is currently used to notify your application that the 5080 link switch is switched away from your application and therefore no graPHIGS API subroutines should be directed to that 5080. This event also indicates that a workstation which is realized as a virtual workstation is disconnected from the actual hardware workstation. The major code contains the workstation's identifier but the minor code is not set.

- 102: Link Switch In

  This class is currently used to notify your application that the 5080 link switch is switched back to the application but, since the state of the 5080 is not known, the workstation should be closed, reopened, and reinitialized. The event is also used to indicate that a workstation realized as a virtual workstation has been reconnected to the actual hardware workstation. The major code contains the workstation's identifier but the minor code is not set.

- 103: Update Completion

  High function workstations can update their display surface without forcing your application to wait for the update to complete. When your application calls an Update Workstation **(GPUPWS)** or Redraw All Structures **(GPRAST)** subroutine, the graPHIGS APIwill initiate the update and return control to your application. Some time later the update will be complete.

  An event of this class is generated when an update completion notification has been requested using the Set Shell Deferral State **(GPSHDF)** subroutine and the graPHIGS APInucleus has completed the processing of a previously invoked **GPUPWS** or **GPRAST** The major code contains the workstation identifier and the minor code contains the display status.

- 104: Lost Input Events

  This event class is returned when the size of a group of simultaneous events exceeds the input buffer size that is being used by the nucleus for sending events from the nucleus to the shell. The major code will contain the workstation identifier and the minor code is undefined.

- 105: Window Resize Notification

  This class is returned when the user changes the size of an X-Window containing the output of a X-Window-type workstation. The major code contains the workstation identifier and the minor code is undefined. In order to receive this event, you must enable notification using the escape **(GPES)** subroutine with function 1009. After receiving a resize notification event, your application can use the Inquire Mapped Display Surface Size **(GPQMDS)** subroutine to obtain the size of the mapped display surface. (The mapped display surface is the subarea of the X-Window that the workstation uses as the display surface for graphics output.) Note that the display surface size returned by Inquire Maximum Display Surface Size **(GPQDS)** and Inquire Maximum Display Surface Size **(GPQADS)**, does not change as the window is resized. You can specify the aspect ratio of the display surface in the X-Window by using the XWINDASP PROCOPT.

201-300=Application Communication Events

As discussed in Chapter 10. Advanced Concepts, your application can be divided into multiple Distributed Application Processes (DAPs) which can communicate through the application message facility supported by the graPHIGS API.

The following class values are defined for application messages:

- 201: Broadcast message

  This class shows that an application message event was generated by another application process using the Send Broadcast Message **(GPSBMS)** subroutine. The major and minor codes and event data are set to the values specified by the other application process.

- 202: Private message

  This class shows that an application message event was generated by another application process using the Send Private Message **(GPSPMS)** subroutine. The major and minor codes and event data are set to the values specified by the other application process.

With the introduction of application message events, the maximum size of events and the number of events that the queue should hold will be more application specific. Therefore, your application may now define the size of the event queue through the External Defaults File (EDF) or through the Application Defaults Interface Block (ADIB) parameter of the Open graPHIGS subroutine. ADIBs and EDFs are discussed in detail in *The graPHIGS Programming Interface: Technical Reference*.

For more information on the use of application messages, see Chapter 10. Advanced Concepts.

301-400=Structure Store Events

This value range is used for events generated by a structure store. The following event class is defined:

- 301: Structure store threshold value exceeded

  In order to prevent loss of data due to out-of-memory errors on a graPHIGS API nucleus, a ″threshold exceeded″ event is supported. The event is generated whenever the storage allocated to a structure store exceeds a specified threshold value. The structure store threshold value is set using the Set Structure Store Threshold **(GPSSTH)** subroutine. The event will only be generated once per setting.

  The event major code contains the structure store identifier that caused the event and the minor code contains the threshold value that was exceeded.

>400=Reserved

This range is reserved for future extensions.

Event Handling

Some applications require greater control over how events are processed. To accommodate that need, the graPHIGS API permits your application to specify an event handling function through the Define Event Handling Function **(GPEVHN)** subroutine. **GPEVHN** takes the following two parameters:

Parameter	Description
handler	the address of your application defined event handler
anchor	a pointer to an anchor block defined by yourapplication.

Whenever an event occurs, your event handler will be called by the graPHIGS API and passed the following parameters by reference using the operating system's standard calling convention:

Parameter	Description
anchor	your application anchor block specified in the call to **GPEVHN**
major	the major code of the event
class	the class of the event
minor	the minor code of the event
length	the length of the event data
data	the event data

Parameter	Description
sflag	an event status flag indicating the following: • whether or not the event queue is about to overflow • whether or not the event queue has already overflowed • whether or not the event data is part of a simultaneous event.
rflag	a return flag which your event handler should set to indicate whether or not the event should be queued or discarded.

Event handlers are subject to the following restrictions:

- Must be written in a programming language which can run without any specific run time environment
- Must follow the standard linkage convention of the operating system on which the graPHIGS API shell is running
- Must not issue any graPHIGS API subroutine calls
- Must not issue any system calls or should not invoke any system function which may require the normal run time environment
- Must return to the graPHIGS API event handler.

The restrictions listed above are intended to prevent an event handler from destroying the normal run time environment. For example, if an event occurs and interrupts the graPHIGS API while processing an API subroutine call, your event handler will get control. If your event handler then calls a graPHIGS API subroutine, the results would be unpredictable.

With the stated restrictions, the function of most event handlers should be limited to the following:

- deciding whether or not the event should be queued based on the data supplied to the event handler
- manipulating the data in the anchor block to communicate with the rest of your application.

Physical Device Support

The phrase ″physical input device″ refers to the actual hardware device used to implement a logical input device as shown in the figure, *PHIGS Input Device Model*. For example, a computer keyboard is a physical input device used to implement a logical string input device.

In order to meet the needs of application programmers, the graPHIGS API extends the PHIGS input model to allow your application program to map additional physical input devices to the PHIGS logical input devices as described in the following sections.

Device Categories

Each physical device is categorized by the type of input it provides. The currently defined categories are `1=BUTTON`, `2=SCALAR`, and `3=2D_VECTOR` as discussed below:

BUTTON Devices: A device in the `BUTTON` category will provide a discrete integer value when some transition of the physical device takes place such as when a key on a keyboard is depressed. The set of integer values that a device may return are not required to lie within one contiguous range. This device is further characterized by the list of all possible integers that it may return. This would correspond to the keys on a keyboard for example. If a device can detect make and break transitions, then distinct integer values are provided for each.

SCALAR Devices: A `SCALAR` device will provide an integer within a single range. A specific `SCALAR` physical device may return either absolute or relative values. For an absolute device, the limits of the range are the devices' physical limits (minimum and maximum value that the device can input). For a relative device, there is no minimum value so a value of zero is used. The maximum value of the range represents the number of increments of physical motion. A unit of physical motion is different for different physical devices. For example, for a dial device, one unit of physical motion equals one turn of the dial.

These values will in reality be related to the operation of the corresponding logical input device. The physical input is mapped to the range of input values that the logical device can input as specified by the application on the initialize device subroutine call.

2D_VECTOR Devices: A 2D_VECTOR device is similar to a SCALAR device except that two integer values are generated by the device instead of one. It is often used to control the cursor displayed on the screen. The device can provide either relative or absolute values. A mouse is an example of a 2D_VECTOR device that would return relative values, while a tablet is a 2D_VECTOR device that would return absolute values. For an absolute device, the limits of the range are the devices' physical limits. The minimum and maximum values generally correspond to the edges of the screen in the case of a device connected to the graphics cursor. For a relative device, there is no minimum limit so a value of zero is used. The maximum value of the range represents the number of increments of one unit of physical motion. For a mouse physical device, one unit of physical motion equals the amount of motion necessary to move the cursor from one edge of the screen to its opposite edge. The limits of the range, as with SCALAR devices, are mapped to the range that the corresponding logical device can input as specified by the application on the initialize device subroutine call.

Note:

1. To determine the range for the physical input device, use the Inquire Physical Device **(GPQPDC)** subroutine.
2. An integer identifier is associated with each physical device to differentiate the devices within each category.

Physical Device Emulation

Some applications require the use of physical input devices other than those defined and explicitly supported by the graPHIGS API In order to meet that requirement and still isolate your application from the characteristics of a physical device and provide the measure mapping operations of logical input devices, the graPHIGS API allows your application to logically and physically detach the default set of physical devices and emulate them instead.

For example, the IBM digitizer is a physical device which may be accessed directly by your application. Tailoring this device to the task the end user is performing will give you desirable results. One possibility is to use a section of the digitizer to emulate the normal operation of the tablet by providing the user with a way to switch between using the tablet or a section of the digitizer as the physical device that is driving the PICK, LOCATOR, and STROKE logical devices. This would allow the user the convenience of using these logical devices from either the tablet or digitizer.

The graPHIGS API cannot provide the same level of functions that your application can by accessing the digitizer directly. Even if the graPHIGS API were to perform the tablet emulation described above, most applications would still require access to the unique characteristics of the digitizer such as control over granularity, access to puck keys, lights, and audible alarms.

This same argument applies to other IBM and non-IBM devices which you may choose to use. Each physical device has many different characteristics and it would be impractical for the graPHIGS API to support every one.

To allow your application to replace and emulate a physical device, the graPHIGS API provides the Set Physical Device Mode **(GPPDMO)** subroutine. The parameters for this subroutine include the physical device category, identifier, and mode. The mode may be specified as either 1=DISABLED or 2=ENABLED When DISABLED, the physical device will not generate any input to the logical input devices to which it is normally connected. ENABLED is the default mode that permits the physical device to drive the logical devices. When a physical device is disabled, your application may emulate it using the Emulate Physical Device **(GPEPD)** subroutine. The parameters of **GPEPD** are defined as follows:

Parameter	Description
wsid	workstation identifier of the device to be emulated

Parameter	Description
category	physical device category (1=BUTTON, 2=SCALAR, or 3=2D_VECTOR)
device	physical device identifier
value	emulated input value(s) to be passed to the logical input device(s).

The simulated input will be passed to the set of logical input devices that are connected to the specified physical device. If the specified input values do not fall within the range normally generated by the physical device or the physical device does not exist, an error will be generated. The normal operation of the logical input device is unchanged.

The characteristics of a physical input device can be determined with the Inquire Physical Device Characteristics **(GPQPDC)** subroutine. The input parameters to **GPQPDC** are the physical device category and a physical device identifier. **GPQPDC** returns information which is dependent on the category of the physical device.

In Version 1 of the graPHIGS API, logical input devices were not available to your application if the required physical input devices were not present. With the introduction of physical input device emulation, there are cases where your application may require emulation of physical devices that are not really present. Therefore, a new defaults option has been defined that allows your application to control whether or not logical input devices are created even if the corresponding physical devices are not present. This option is specified as a 'procopt' in either an External Defaults File (EDF) or through a parameter of the Create Workstation **(GPCRWS)** subroutine. When logical devices are specified unconditional upon creation, your application must use the Inquire Physical Device Characteristics **(GPQPDC)** subroutine to determine if the physical device is actually present.

Input Trigger Extensions

Each logical input device may have one or more *trigger levels* with one trigger each. Each trigger level corresponds to an atomic operation on the measure or state of an input device.

The first or *primary*trigger level causes the logical input device to pass its current measure value back to your application or to add the measure to the event queue depending on the mode of the logical input device. Every logical input device has a primary trigger.

Trigger levels greater than the first (*secondary*) are used to change the measure or the state of the logical input device. As an example, the following table summarizes the secondary triggers for the STROKE and PICK logical input devices supported by the 5080 workstation type:

Table: 5080 Secondary Trigger Levels

Device	Trigger Levels
STROKE #1	**1 -** Ends the accumulation of points into the stroke buffer. **2 -** Initiates the accumulation of stroke points into the stroke buffer.

Device	Trigger Levels
STROKE #2	**1 -** Moves the editing position in the stroke buffer backward one point. **2 -** Moves the editing position in the stroke buffer forward one point. **3 -** Inserts a point into the stroke buffer. **4 -** Deletes the current point from the stroke buffer.
PICK #1	**1 -** Suppresses the pick correlation process. **2 -** Activates the pick correlation process.

A trigger for any level can be generated by one or more physical input devices in conjunction with a *trigger mapping* A trigger only affects a logical input device that is in `EVENT` mode or has a `REQUEST` pending. Triggers at any other time have no effect on the logical input device.

The relationship between the physical and logical input devices is determined by the workstation developer and cannot be altered. For example, the following table lists the physical input devices and the logical input device measure and trigger processes which use them:

Table: 5080 Physical/Logical Device Correspondence

Physical Device	Logical Device Measure	Logical Device Trigger
VALIGN=″TOP″>Tablet	LOCATOR 1,2 STROKE 1,2 PICK 1	
Puck or Stylus	CHOICE 2	LOCATOR 1,2 STROKE 1,2 VALUATOR 1-8 PICK 1 STRING 1-16
LPFK	CHOICE 1	LOCATOR 1,2 STROKE 1,2 VALUATOR 1-8 PICK 1 STRING 1-16
Keyboard	CHOICE 3 CHOICE 4 STRING 1-16	LOCATOR 1,2 STROKE 1,2 VALUATOR 1-8 PICK 1 STRING 1-16
Dial 1	VALUATOR 1	VALUATOR 1
Dial 2	VALUATOR 2	VALUATOR 2
Dial 3	VALUATOR 3	VALUATOR 3
Dial 4	VALUATOR 4	VALUATOR 4
Dial 5	VALUATOR 5	VALUATOR 5
Dial 6	VALUATOR 6	VALUATOR 6

Physical Device	Logical Device Measure	Logical Device Trigger
Dial 7	VALUATOR 7	VALUATOR 7
Dial 8	VALUATOR 8	VALUATOR 8

Each logical input device has a set of BUTTON physical devices which are used to trigger any of the logical device's trigger levels. Since a BUTTON physical device can generate many values, the trigger mapping for each logical input device acts as a filter on these values so that only a subset will actually trigger the logical device.

The programmable trigger concept allows your application to modify the trigger mapping by selecting which buttons on a physical BUTTON device will trigger each of the trigger levels of the logical device. A trigger mapping includes the following information:

Trigger Type A trigger type value greater than zero represents a physical device identifier from the BUTTON category that is to be used to generate the trigger. Values less than zero are reserved for triggers not generated by physical devices.

Trigger Qualifier A trigger qualifier defines which values from the physical BUTTON device will trigger the specified trigger level. These values may be specified as ranges for convenience.

The list of BUTTON type physical devices that may be used for triggering a specific logical input device can be determined using the Inquire Input Trigger Capabilities **(GPQIT)** subroutine. This subroutine defines the correspondence between the physical devices and the triggers of the logical input devices.

Warning: Temporary Level 4 Header

National Language Pre-editing Mode Effect on Triggers: Some national languages such as Japanese allow the user to enter a pre-editing mode or to view pop-up translation menus (auxiliary windows) while typing in a string device. In these modes, characters are interpreted by a special language processor (the Input Method), before they are entered into the graPHIGS string device.

Triggers behave as follows during pre-editing mode and when auxiliary windows are present:

- BUTTON physical device 4 (the keyboard) does not provide input to any measure process nor to any trigger process. For example, during pre-editing mode, your application will not receive any choice device 4 input events.
- A STRING device in pre-editing mode or with auxiliary windows present cannot be triggered by any trigger. For example, a Kanji string device in pre-editing mode cannot be triggered by a mouse button.

Change in Measure Trigger

Programmable triggers have been extended to include the notion of firing the primary trigger when the physical measure changes. The change in measure trigger allows your application to follow the motion the mouse or tablet puck as it moves. This is a very common mode of operation. For example, your application may use the cursor to input values for a transformation matrix based on cursor's XY screen location. This type of trigger has always been used for the VALUATOR devices in EVENT mode but was not exposed since the VALUATOR does not have programmable triggers.

The change in measure trigger is set by the Set Input Device Trigger **(GPIT)** subroutine where the trigger type parameter is -1 and is only valid for the primary trigger level of a logical input device.

The low trigger threshold qualifier specifies the minimum change, in 2D_VECTOR physical device units, needed for triggering. Use **GPQPDC** to determine the valid ranges for the 2D_VECTOR devices. The high trigger qualifier is not used for this trigger type and must be set to 0.

For PICK devices, the graPHIGS API accumulates the distance from one event to the next and correlates only when the low qualifier threshold is exceeded, except for the last event occurring in a chain of events. The last in a chain of events is always correlated, regardless of the low qualifier. Events accumulated prior to the last event are also correlated when they exceed the low qualifier threshold.

This trigger type can be interpreted as an option for the trigger mapping process of the logical input device. When this trigger type is used, the trigger mapping process monitors the physical measure for change and when the cumulative change is more than the value contained in the low trigger qualifier, the primary trigger is fired.

Use the Inquire Input Trigger Capabilities **(GPQIT)** subroutine to determine whether an input device supports this trigger type.

Note: The graPHIGS API does not limit the change in measure trigger type to use by Distributed Application Processes (DAPs) but the communications overhead will be high if the program receiving the events does not reside in the same node as the one to which the user's display is directly attached.

Trigger When Primary Fires

The *trigger when primary fires* mode allows the secondary trigger to correspond to the same event as the primary trigger. Trigger when primary fires is set by the Set Input Device Trigger **(GPIT)** subroutine where the trigger type parameter is -2 and is only valid for the secondary trigger level of a logical input device. The low and high trigger qualifiers have no meaning and must be set to 0.

When used in combination, *trigger by measure* (trigger type -1), and *trigger when primary fires* (trigger type -2), implement a *sliding pick* device that returns an event for the first primitive crossed if correlation is set to ON *trigger by measure* should be set to a small value to avoid skipping primitives.

Input Device Dependent Triggers

Input Device Dependent Triggers are used by some logical input devices as a default trigger type because the input device uses a special type of trigger or one which is mode dependent. A VALUATOR input device on the 5080 workstation is a good example of this. On a 5080 workstation the enter key is the primary trigger for VALUATOR devices in REQUEST mode but movement is the primary trigger in EVENT mode. The default trigger for a VALUATOR is "logical input device dependent" to allow workstations to support programmable triggers without returning ambiguous or incorrect information on inquiry subroutines. The Input Device Dependent Trigger type is only returned from inquiry subroutine and is never valid as a parameter to a trigger setting subroutine call.

Input Device Enhancements

Cursor Shape Definition

On several of the workstations supported by the graPHIGS API, the graphics cursor is used as the prompt for LOCATOR, STROKE, and PICK input devices. Each one of the logical input devices has a different prompt shape. The LOCATOR uses a "+" shape, the STROKE uses an "X" and the PICK device uses a box whose size corresponds to the current pick aperture. The prompt is only displayed when the device is active and the cursor is in the input device's ECHO area. Since ECHO areas may partially overlay each other, the shape of the cursor may change as it is moved around the screen passing through different combinations of active ECHO areas.

The cursor is drawn by display adapter hardware in some configurations. This produces cursor movement performance that is independent of screen content. Many application developers like to define their own cursor shapes. Two workstation subroutines are provided to accomplish this. The Set Cursor Shape **(GPCUS)** subroutine selects the cursor shape for a specified workstation. The second parameter of **GPCUS** is an index which selects either a specific type of cursor, such as a cross hair, or an entry in the workstation's cursor shape table. **GPCUS** also allows the selection of a two-color (XOR echo) representation of any of the workstation cursors, as well as an option to disable the graPHIGS API cursors

and allow the application to use the default cursor facilities native to a window system such as those provided by X-Windows. (See HWCURS PROCOPT for information on using the hardware crosshair cursor.) Use the Set Cursor Representation **(GPCUR)** subroutine to redefine an entry in the workstation's cursor shape table. When an entry is selected, that cursor shape is used for all logical input devices that require a graphics cursor. The cursor shape, therefore, is not unique to the logical input device nor does it change as the cursor moves around the screen unless it moves outside the echo area or the application changes it. **GPCUR** takes the following parameters:

Workstation Identifier
Specifies the workstation containing the cursor shape table that is to be set.

Cursor Shape Table Index
Specifies the entry of the table to be set.

Cursor Format
Specifies the format of the cursor definition.

Cursor Definition
Contains the data to define the cursor.

The size of the cursor shape table and the cursor formats that are supported by a workstation can be determined through the Inquire Cursor Facilities **(GPQCUF)** subroutine. The only cursor format currently defined requires a two-dimensional array of pixel values with 1 bit per pixel. For this format, the cursor definition must contain the following:

- The size of the pixel array (number of rows and columns)

  Note: For this format, the specified array sizes must match the maximum size pixel array which can be inquired using **GPQCUF**
- The pixel array which is stored in row major order with 8 pixels packed per byte
- Each row must start a 32-bit boundary.

The hot point of the cursor is always the center of the pixel array.

PICK Device Enhancements

Pick Selection Criteria: Based on the organization of your graphics data and performance considerations, you may choose to have the graPHIGS API return the `FIRST` or `LAST` primitive which is detected by a `PICK` device. In addition, some application programmers would like their applications to receive every primitive that passes through the `PICK` aperture. This allows your application or the user to make the decision as to which primitive of those passing through the `PICK` aperture is the actual target of the operation. You may also choose to perform some operations on every primitive which passes through the `PICK` aperture.

The pick selection criteria controls whether the `FIRST`, `LAST` or `ALL` primitives passing through the `PICK` aperture are returned to your application. This characteristic of a `PICK` device can be initialized through the Set Pick Selection Criteria **(GPPKSC)** escape subroutine. One of the parameters passed to **GPPKSC** controls whether or not the pick selection criteria is `FIRST`, `LAST`, or `ALL` For selection criteria `ALL`, every primitive which intersects the `PICK` aperture will be echoed when pick echoing is `ON`.

If the Pick Selection Criteria is set to `ALL` and a `REQUEST` or `SAMPLE` function is invoked for the `PICK` device, only information about the `FIRST` pick primitive will be returned. When the `PICK` device is in `EVENT` mode, the list of picked primitives will be returned as simultaneous events. Each event will only contain the information for a single primitive. Several locations on the same curve or surface may lie within the `PICK` aperture. Each primitive will be returned only once, even if the selection criteria is set to `ALL`.

In addition to controlling the picked primitive using the `FIRST`, `LAST`, and `ALL` order of processing, your application can control the picking of primitives effected by HLHSR processing. The criteria `FIRST_VISIBLE`, `LAST_VISIBLE`, and `ALL_VISIBLE` refer to those primitives that are in the `PICK` aperture and are visible

based on the HLHSR processing. The criteria FIRST, LAST, and ALL refer to those primitives that are in the PICK aperture but not necessarily visible based on HLHSR processing. Any effect due to HLHSR processing does not change which object is picked (i.e., the selected object may not be displayed when HLHSR is active). If a workstation does not support HLHSR processing, then the visible PICK modes operate the same as normal PICK modes.

Initial Correlation State: PICK correlation is the process of determining which graphical elements are intersected by a PICK device's aperture. Typically, correlation is turned on when the user causes a secondary trigger to fire by pressing and holding down a mouse, stylus, or puck button. Releasing the button causes the PICK device's primary trigger to fire and the PICK measure to be sent to your application. Some applications developers however, always want PICK correlation turned ON.

By default, the graPHIGS API defines PICK correlation to be OFF when a PICK device is activated. If you want correlation always ON then the PICK device triggers must be set so that correlation is turned ON by any trigger and only turned OFF if the device is put in REQUEST mode (deactivated). This is not convenient since the user must fire a trigger (press a button) to start the correlation process each time the PICK device is activated.

To alleviate this inconvenience the Set Initial Pick Correlation State **(GPIPKC)** escape subroutine is defined. **GPIPKC** takes the following three parameters:

Parameter	Description
wsid	the workstation identifier of the PICK device whose correlation state is to be changed
device	the PICK device identifier of the PICK device whose correlation state is to be changed
state	the Pick Correlation State (1=OFF or 2=ON)

A Pick Correlation State of ON specifies that when the device is activated, correlation will be turned on immediately. The default value of OFF specifies that correlation will start when the PICK device's secondary trigger is fired.

Extended Pick Input Data: Some applications require more information about a picked primitive than a standard PICK device will return. For example, an application may require the view identifier of the picked primitive. Therefore, the PICK device measure has been extended in the graPHIGS API to include the following information:

Label Identifiers
Each level of the pick path hierarchy includes the following:

- Structure identifier
- Pick identifier
- Label identifier
- Element number

View Index
Specifies the view table index that was used to render the picked primitive.

Composite Modeling Matrix
Specifies the concatenation of the local and global modeling transformation matrices that were used to render the primitive.

Aperture Location
Specifies the location of the center of the PICK aperture in Normalized Projection Coordinates (NPC).

The extended PICK information is returned by the Get Extended Pick **(GPGTXP)**, Request Extended Pick **(GPRQXP)**, and Sample Extended Pick **(GPSMXP)** subroutines. If any of the existing pick input subroutines (Get Pick **(GPGTPK)**, Request Pick **(GPRQPK)**, and Sample Pick (**GPSMPK**)) are issued to a PICK device which returns extended information, only the current pick path without labels will be returned.

Some workstation platforms do not support the extended PICK class. Therefore, to determine if a workstation supports extended PICK, use the Inquire Pick Measure Type **(GPQPKT)** subroutine. If one of the extended PICK input subroutines is called for a device which does not return extended PICK information, the subroutine call will be rejected with an error.

Chapter 19. Fonts

This chapter describes the treatment of fonts by the graPHIGS API It includes a discussion of font files, font directories, and font inquiry subroutines.

Font Files

In the graPHIGS API, the shell and nucleus may be running as two different processes. In some cases, they may be running on two different machines with two completely different disk systems. The font definition is read from the disk system which the nucleus can access. When the shell receives a call to the Activate Font **(GPACFO)** subroutine, it sends a corresponding request to a nucleus which owns the specified workstation. The nucleus searches its own disk system for the specified character set/font pair and loads it into the specified workstation.

The font file is taken from the nucleus's disk to:

- **Minimize the data transmission (I/O) for the font activation.** Because workstations are nucleus resources, the font data is taken from the resources belonging to the nucleus.
- **Enable multiple application processes to share the same font definition.** Even when a nucleus is used by multiple applications which are running on different nodes, the font definition is taken from the same place and is shared by all applications processes.
- **Keep the meaning of the character set identifier and font number consistent.** A given character set and font pair is always mapped to the same file independent of where and in what environment your application is running. The mapping is determined only by the environment where the nucleus is running.

Font Directory

Though the font files available to a nucleus will be enough for many applications, there are some cases where you may want to load a font definition to a workstation from the shell's disk. Following are three examples:

1. The font is an application defined font and you do not want to install the font definition onto the nucleus's disk. If all font definitions must reside on the nucleus's disk, there is no way to avoid exposing your font definition to other applications.
2. If your application uses multiple nuclei, either simultaneously or one at a time, you may not want to install a font definition to each nucleus. Especially if the font definition may be changed or re-defined, then you might want to keep the font definition file on your application's disk to make it easy to maintain.
3. The nucleus does not have enough disk space to install all font definitions that your application requires. This will occur when the nucleus is implemented on a workstation with a small disk system or or one without any disk system such as the IBM 6090.

As a result of these requirements, the graPHIGS API supports the concept of a font directory. The font directory can be viewed as a run time disk system for fonts, which is dynamically created and attached to the shell by the Create Font Directory **(GPCRFD)** subroutine. A font directory is a nucleus resource and can be shared by multiple application processes using the Attach Resource **(GPATR)** subroutine.

When a font directory is created, it is empty. Your application can load any character set/font pair to the font directory using the Load Font **(GPLDFO)** subroutine. When the shell receives this call, it searches its own disk system for the specified font definition. When the shell finds the font file, it reads the file and sends it to the specified font directory. If the specified character set/font pair is already in the font directory, the font is replaced.

A given workstation can have only one font directory associated to the workstation at any given time. This association is set using the Associate Font Directory with Workstation **(GPAFDW)** subroutine. If the workstation already has a font directory associated with it, the subroutine call replaces the currently associated font directory.

When Activate Font **(GPACFO)** subroutine is called, the following actions are performed by the nucleus:

1. The nucleus checks the workstation to see if it has an associated font directory. If so, it then searches for the specified font definition. If the font definition is found, it is activated on the workstation.
2. If the specified font definition does not exist in the font directory, or the workstation has no associated font directory, the nucleus' disk system is searched, and if the font exists, the font is activated.
3. The nucleus generates an error if the specified font definition does not exist in an associated font directory or in the nucleus' disk system.

As described above, the search of a font directory associated to a workstation supersedes the normal font file search order in the nucleus. Note that the contents of an associated font directory affects only subsequent Activate Font subroutines. The contents of a font directory does not affect any character set/font pairs already active on the workstation. Conceptually, each font definition is copied from the font directory to the workstation's font pool when it is activated and becomes independent of the font directory. Even if the font directory is disassociated from the workstation, character set/font pairs which are currently active on the workstation remain active.

Font Inquiries

In Version 1 of the graPHIGS API, the Inquire Annotation Font Characteristics **(GPQAFC)** returned the following information about how a workstation would process a specific character set and font pair:

- The highest text precision available
- The nominal character height
- The number of character height scale factors supported
- A list of the height scale factors supported
- For each height scale factor, the number of expansion factors and the minimum and maximum expansion factors.

The expansion factor information was returned as a deviation from the nominal aspect ratio of the corresponding geometric text font.

The meaning of the **GPQAFC** inquiry has been changed slightly in Version 2 but will still produce compatible results for Version 1 applications. The expansion factor information has been redefined to be relative to the annotation nominal font aspect ratio instead of the geometric font aspect ratio. To provide compatibility, all the annotation fonts will have the same aspect ratio as the geometric font with the same character set and font identifiers.

The Inquire Extended Annotation Font Characteristics **(GPQXAF)** subroutine will return the following information in addition to the information returned by **GPQAFC**:

- Whether the font definition includes proportional character information
- The amount of space between the capline and topline as a fraction of the character height
- The amount of space between the base and bottom lines as a fraction of the character height
- The nominal aspect ratio of the font.

Unlike geometric text fonts, annotation text fonts are workstation specific.

With the introduction of the shell-nucleus concept and the associated nucleus resources, a font may reside in many places. These repositories for fonts include the shell font storage, the nucleus font storage, font

directories, and workstations. A given character set and font number could have a different definition in each location. To inquire information about a font in the shell's font storage, use the following inquiry subroutines:

- Inquire Font Characteristics **(GPQFCH)**
- Inquire Font Aspect Ratios **(GPQFAR)**

The primary use of **GPQFCH** and **GPQFAR** is to compute the length of a text string. **GPQFCH** returns whether the font has character box definitions for each character (proportional positioning), the amount of space between the capline and top line, the amount of space between the base and bottom lines and the aspect ratio of the nominal character box. **GPQFAR** returns the aspect ratio of each character within a specified string. This is only useful for a font which has proportional positioning information.

Chapter 20. Images

This chapter describes the basic concepts related to the display of image data by the graPHIGS API.

An image is defined by an image board, which is a two-dimensional (2D) array of data values treated as a resource of the graPHIGS API nucleus. Note that an image board is not a physical piece of hardware, but a conceptual place to store image data. An image can be displayed on the workstation by mapping a defined image to a workstation view.

Image Model

In the most general sense, an image is a mapping function from a 2D surface into a color space. To represent the function in a digital form, the image is sampled both within its definition domain and value domain. We assume that the definition domain is a planar rectangle and the function is sampled at grid points with regular intervals. Therefore, the image is represented by a 2D array of color values.

Color values can be represented by a vector within a normalized three-dimensional color space, so the image can be represented by a two-dimensional array of three-dimensional vectors. However, such a representation requires a large amount of storage when each color value is represented by a triplet of real values. To reduce the amount of data, color values are quantized. Two quantization methods are commonly used in many image applications.

One method is to quantize three color primaries independently and is called *scalar quantization*, as shown in the following figure. The normal RGB representation of images is a typical example of this method. Each pixel is represented by three indexes each of which specifies one representative value of a color component. Note that the quantizations of three primaries need not be equal interval sampling (for example, r-corrected RGB) nor to be identical depth (for example, the broadcast TV color system, digital YIQ, with different bit length).

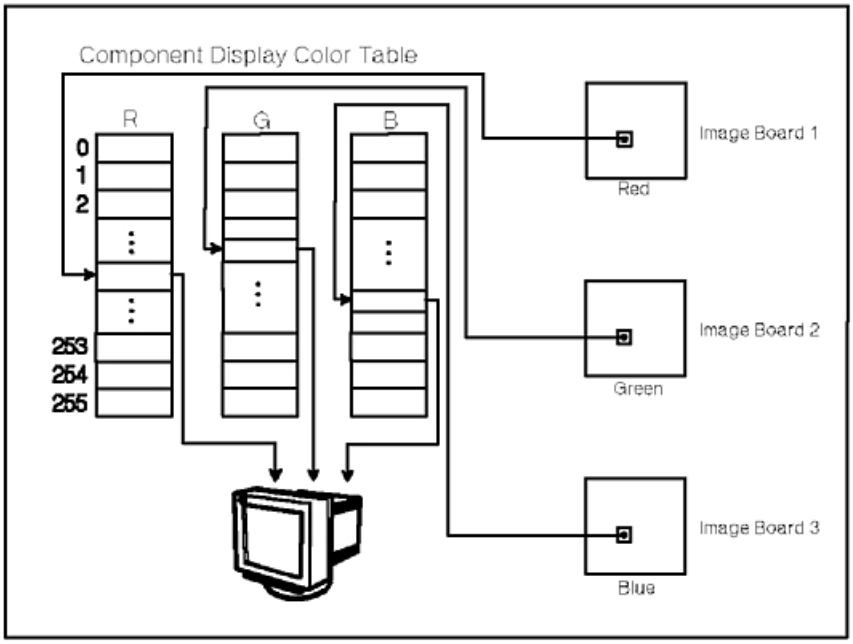


Figure 94. Scalar Quantization. This diagram shows the quantization method in which three color primaries are quantized independently.

Another method is to quantize color values as three-dimensional vectors and is called *vector quantization*, as shown in the following figure. An image handled by the Pixel **(GPPXL2** and **GPPXL3)** subroutines of the API is a typical example of this method. Each pixel is represented by an index specifying one representative color selected from a three-dimensional color space. Note that this method of quantization includes binary and grey scale images as special cases in which the color representatives are selected from monochrome colors.

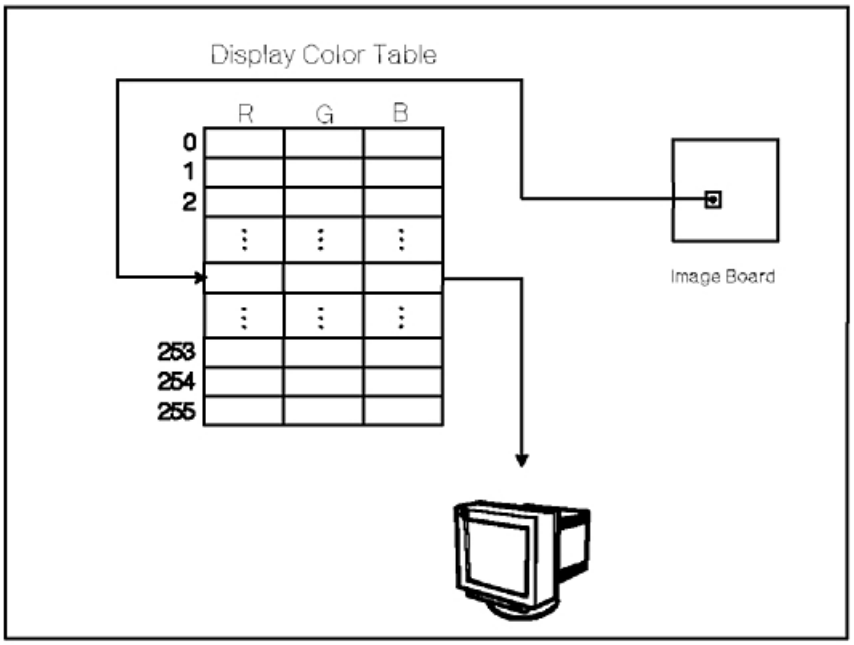


Figure 95. Vector Quantization. This diagram shows the quantization method in which color values are quantized as three-dimensional vectors.

Notice that in both cases, an image is represented by a 2D array of color table indexes. The only difference between them is the connection between the array and color table. In the scalar quantized image, the color table is indexed by three indexes separately for each color component but in the vector quantized image, it is indexed as a whole by a single index.

Defining, manipulating and displaying an image is similar to that of geometric objects. In addition, the display of an image merges into the rendering pipeline of geometric objects, resulting in both the image and geometric data showing on the display surface. You may display an image using of the following graPHIGS API facilities:

- Image Board - An area of storage that holds the quantized image data
- Image Board Manipulation - graPHIGS subroutines that allow you to create and modify the contents of the image board
- Image Color Tables - Color tables associated with image boards that are used to obtain the color value of a pixel
- Image Display - Subroutines that define the location of the image in World Coordinates

These facilities are discussed below.

Image Board

To store the 2D arrays of indexes described above, the graPHIGS API supports a nucleus resource called an *image board*. Each pixel is referred to by two indexes where the pixel at the lower-left corner has an index pair (0,0). Pixels at the lower-right, top-left and top-right corner are referred to by index pairs (SX-1,0), (0,SY-1) and (SX-1,SY-1), respectively, where SX and SY are the horizontal and vertical size of the image board.

Conceptually, the image board is an abstraction of a frame buffer for raster type displays. However, image boards are not necessarily realized by special memory such as a physical frame buffer. A nucleus implementation can utilize any storage for an image board and your application can create and delete image boards independent of the physical frame buffer.

An image board is created using the Create Image Board **(GPCRIB)** subroutine. **GPCRIB** takes the following parameters:

- Image board identifier
- Nucleus identifier
- Bit depth
- Horizontal and vertical size
- Image board type
- Type dependent image board description

The bit depth parameter must be one of those supported on the target nucleus. The graPHIGS API generally supports bit depth of 1,2,4,8 and 12, but the application should inquire the actual list of bit depths available on a given nucleus using the Inquire Image Board Facilities **(GPQIBF)** subroutine.

The conceptual size of the image board is determined by three size parameters, bit length of each pixel, and the horizontal and vertical dimensions of the image board array. However, the actual amount of storage used for an image board may differ between nucleus implementations because of the differences in their storage organization. A nucleus implementation can use any storage mechanism (for example, it can use any compressed format) to realize image boards.

You can attach an image board created by another application process to your application using the Attach Resource **(GPATR)** subroutine with the following parameters:

- Resource type (`3=IMAGE_BOARD`)
- Image board identifier
- Nucleus identifier
- Resource identifier
- Password

After the image board is attached to your application's shell, you application can access it as though it created the image board itself.

As for other nucleus resources like structure stores or workstations, the resource identifier of the image board and its password is known only to the application process which created the image board. Therefore, your application process which uses the Attach Resource **(GPATR)** subroutine, for an image board, must get the information from the application process which created the image board.

An image board is detached from the shell using the Detach Resource **(GPDTR)** subroutine which takes with an image board identifier as a parameter.

Manipulation of Image Board Content

The graPHIGS API provides functions that enable you to manipulate image board data content.

The Fill Rectangle **(GPFRCT)** subroutine is used to fill a portion of an image board with a constant value. Note that the contents of an image board are undefined when the image board is created. If the application wants to initialize all pixels of the image board to a constant value (clear the image board), the application must explicitly fill the entire image board with the value by using this subroutine.

Write Rectangle **(GPWRCT)** and Read Rectangle **(GPRRCT)** subroutines are used to move image data to and from an image board. Both subroutines take the following parameters:

- An identifier of an image board to which pixel data is written
- Two integers specifying the location of the low-left corner of a rectangle in the image board
- Two integers specifying the size of pixel data to be moved
- An integer specifying the type of image format and additional parameters which depend on the type
- An application's storage area which contains the image data
- Two integers specifying the location of the low-left corner of a rectangle in the application storage area

The graPHIGS API supports one type of image format: the pixel array. This type of image format description requires the following information:

- An integer specifying the bit length of each pixel passed in the image data parameter. The graPHIGS API supports bit depth of 1,2,4,8 and 16.
- An integer specifying the number of pixels in a row (horizontal size).
- An integer specifying the number of rows.
- An integer specifying the order of pixels in the pixel buffer.
  - `1=LEFT_TO_RIGHT_BOTTOM_TO_TOP`
  - `2=LEFT_TO_RIGHT_TOP_TO_BOTTOM`

The table below lists the minimum and recommended horizontal size based on the image format depth.

Bit Depth	Required Minimum Horizontal Size	Recommended Horizontal Size
1	Multiple of 8	Multiple of 32
2	Multiple of 4	Multiple of 16
4	Multiple of 2	Multiple of 8
8	No Restriction	Multiple of 4
16	No Restriction	Multiple of 2

Your application can also transfer image data from one image board to another using the Transfer Rectangle **(GPTRCT)** subroutine.

In addition, your application can perform operations on one or two image boards and place the result into another image board using the following two subroutines:

- Two Operand Pixel Operation **(GPTWPO)**
- Three Operand Pixel Operation **(GPTHPO)**

GPTWPO processes a rectangular area of the source image board and places the result into another rectangular area of the target image board. The operation type parameter specifies the process to be performed on the source rectangle. The following operation is defined:

- Reflection (change pixel order)

GPTHPO performs a binary operation on the two source image boards and stores the result into the target image board. The following two operations are currently defined:

- Logical operations
- Arithmetic operations

Image boards which are the target of this subroutine call must reside on the same nucleus and the actually supported pixel operations may depend on the nucleus where these image boards reside. Your application can inquire a list of operations supported by a given nucleus with the Inquire Available Pixel Operations **(GPQPO)** subroutine.

All source and target rectangles must always have the same horizontal and vertical size and be entirely within the source image data. If the target rectangle is not entirely within the target image data, pixels falling outside of the target are discarded. Also, the bit lengths of the image boards need not be the same. When the bit length of the target pixel is less than that of the source, the source bit string is truncated by discarding the most significant part. When the bit length of the target pixel is larger than that of the source, the source bit string is expanded by adding 0-bits to the most significant part. Note too, that the target image board for **GPTRCT**, **GPTWPO**, and **GPTHPO** can be the same functions as the source rectangles.

Image Color Table Connection

To display the contents of an image board, your application must call the Define Image **(GPDFI)** subroutine with the following parameters:

- Workstation identifier
- Image index
- Image connection type
- Number of image boards
- A list of image board identifiers

An image definition is stored in the specified entry of the workstation's image table. Each workstation has an image table with a predefined number of entries and all entries are initialized as undefined when the workstation is opened.

The image connection type specifies the relationship (″connection″) between the image board contents and the color table. This defines the colors of pixels. (These colors are then quantized and written to the frame buffer.) There are 3 types of connections supported:

- COMPONENT

  The component connection is used for scalar color quantization. Three image board identifiers are specified. To obtain a pixel color, three indices are used, one from each of the three image boards at the same pixel location. The three indices are used independently to select a color component from the specified color table. The first index selects the first color component, the second index selects the second color component, and the third index selects the third color component. The resulting color value is the color of the pixel, as shown in the following figure:

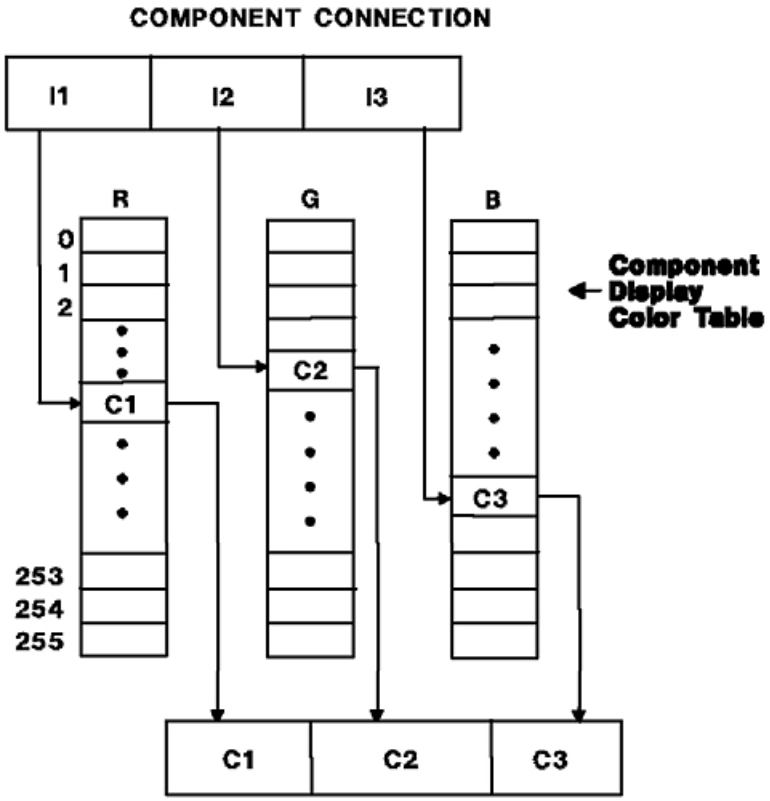


Figure 96. Component Image Board Connection. This diagram shows that the resulting color value is the color of the pixel.

- INDEXED

  The indexed connection is used for vector color quantization. One image board identifier is specified. A pixel color is obtained by using the index at the pixel location. The index is used to select the three color components from the specified color table. The color value obtained is the color of the pixel.
- FRAME-BUFFER-COMPATIBLE

  The frame-buffer-compatible connection is used for a color connection that uses the workstation's color facilities. The number of image board identifiers specified is the same as the number of frame buffer components of the workstation. (Obtain this value using the Inquire Frame Buffer Characteristics **(GPQFBC)** subroutine.) In addition, the color table used will always be the default color table. If the frame buffer has 1 component, then frame-buffer-compatible is similar to an indexed connection using the default color table. If the frame buffer has 3 components, then frame-buffer-compatible is similar to a component connection using the default color table.

The actual meaning of the Frame Buffer Compatible connection type is that operations on the image data are performed in pixel space, not in a color space. In many cases, this will be the preferred connection type for workstations with a component frame buffer since the pixels values themselves can be treated as a color space. No additional color mapping is necessary.

An entry of the image table becomes undefined when the Cancel Image **(GPCAI)** subroutine is called with a specified image index.

Your application can inquire the image capabilities of a workstation and the current status of an image definition using the subroutines:

- Inquire Image Definition Facilities **(GPQIDF)**
- Inquire Image Board Characteristic **(GPQIBC)**

Your application can create color tables that are used during the image color processing using the Create Color Table **(GPCRC)** subroutine. These image color tables are identified by a color table identifier, which must be a positive integer. (A workstation has two color tables: a `DISPLAY` color table, whose color table identifier is the value -1 and a `RENDERING` color table, whose color table identifier is the value 0.) The Set Extended Color Representation **(GPXCR)** subroutine sets the image color table entries. When using a `COMPONENT` or `INDEXED` image connection type, you must pass the color table identifier of the image color table to be used to obtain the color value of each pixel.

To delete an image color table, use the the Delete Color Table **(GPDLC)** subroutine, which takes an image color table identifier as a parameter. If an image is currently defined on the workstation using this color table, then the image is undefined (cancelled), before the color table is deleted.

Image Display

You can display any defined image by mapping a rectangular part of the image onto a parallelogram in the World Coordinates (WC) using the following subroutines:

- Create Image Mapping 2 **(GPCIM2)**
- Create Image Mapping 3 **(GPCIM3)**

The mapped image is identified by an image mapping identifier specified by the application. The **GPCIM2** and **GPCIM3** subroutines require a workstation identifier, view index, image mapping identifier, image index, image rectangle, three points in WC, mapping type and a display priority.

The parallelogram in the WC space onto which the image rectangle is mapped is defined as follows. Let P, Q, and R be three points specified throughout either **GPCIM2** or **GPCIM3** The parallelogram has its four vertices on P,Q,R and Q+(R-P). Image data within the image rectangle is mapped to this parallelogram such that the bottom-left corner is on P, bottom-right corner is on Q and top-left corner is on R.

The actual appearance of an image when it is displayed on a display surface is controlled by the mapping type parameter of **GPCIM2** and **GPCIM3**. Currently only one mapping type, `PIXEL_BY_PIXEL`, is defined. In `PIXEL_BY_PIXEL` mapping, only the first point P the image mapping is transformed by the viewing transformation and workstation transformation. Pixel data within the image rectangle is displayed without any transformation. The view clip and workstation clip may be applied to the image in a workstation-dependent way. This display method corresponds to the pixel primitives supported by the graPHIGS API.

Although images are mapped in 3D World Coordinates as are graphical primitives, images are not treated as graphical entities. All images within a view are always rendered as a background picture of the view and no hidden line or hidden surface process (HLHSR) is applied to the images. The priority parameter of **GPCIM2** and **GPCIM3** controls only the priorities of images and has no relation to those of graphical root structures. Also, image data is not pick detectable. An image mapping can be performed in a view in `PARALLEL` projection only. If the view is in `PERSPECTIVE` projection, an error is generated. If a `PARALLEL` projection is changed to `PERSPECTIVE` projection, then the image mapping is deleted.

When **GPCIM2** or **GPCIM3** is applied to an already existing image mapping, it is treated as a modification of the image mapping parameters. When an image definition which is already used for any image mapping is re-defined, the newly defined image is displayed at the next update.

An image mapping is deleted from the view by the Delete Image Mapping **(GPDIM)** subroutine or when the image definition referred to by the image mapping becomes undefined. Any modification of the image definition other than cancelling the image, does not affect any image mappings. Such modification will be visualized at the next update.

Inquire the workstation capabilities related to the image mapping and current status of the image mapping using the following graPHIGS API subroutines:

- Inquire Image Definition Facilities **(GPQIDF)**
- Inquire Image Mapping On Workstation **(GPQIMW)**
- Inquire Image Mapping On View **(GPQIMV)**
- Inquire Image Mapping of Image **(GPQIMI)**
- Inquire Image Mapping Characteristic **(GPQIMC)**

Chapter 21. Error Handling

This chapter describes the error handling facilities of the graPHIGS API These facilities give your application the ability to perform the following:

- Let the graPHIGS API log errors to a specified file as they are detected (the default)
- Turn off parameter checking performed by the graPHIGS API shell
- Turn on and off the processing of errors
- Set up a first-level error handler to receive control each time an error is processed
- Set up a second-level error handler to receive control based on the severity of the error as assessed by the first level error handler

The following two figures depict the sequence of events that occur when the graPHIGS API detects an error. Error detection and the processing of errors is discussed in detail in the following sections.

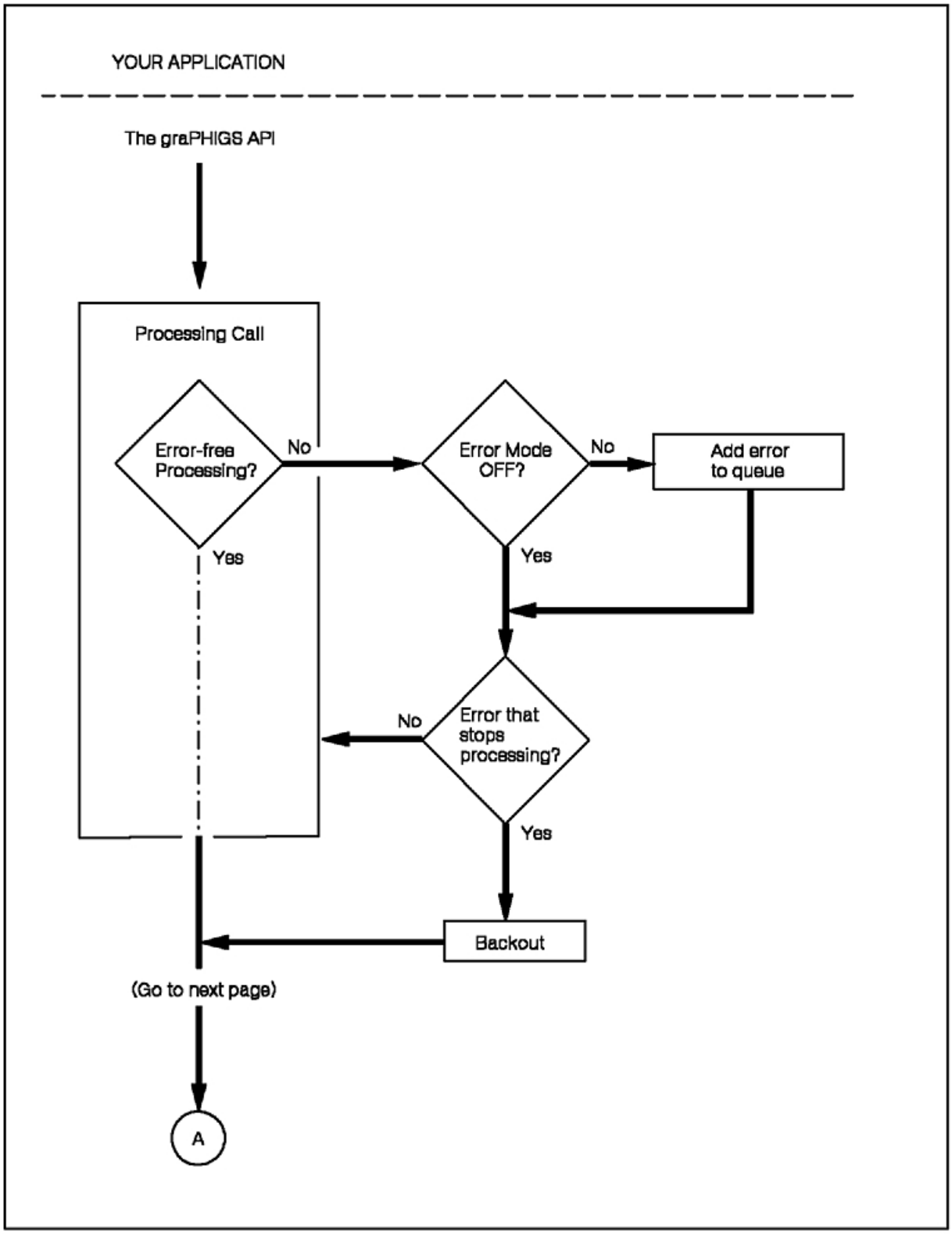


Figure 97. Error Handling Flow of Control (Part 1 of 2). This flow chart shows the sequence of events that occur when the graPHIGS API detects an error. If there is no error-free processing and error mode is not off, then the error is added to the queue. The API checks to see if this is an error that stops processing. If it is not, then the API continues to point A. If it is, then the API backs out. If the error mode is off, then the API goes on to check if this is an error that stops processing as described above.

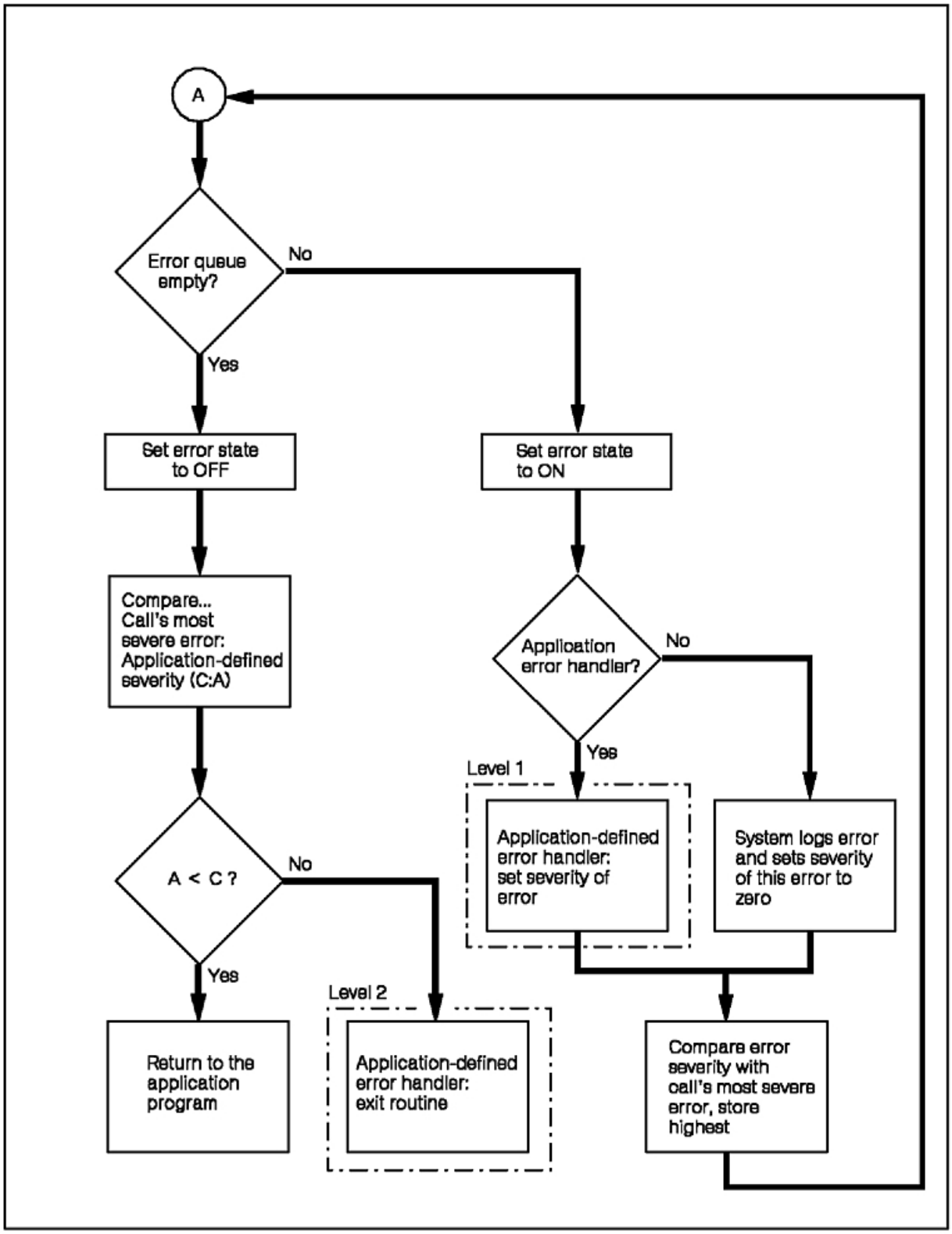


Figure 98. Error Handling Flow of Control (Part 2 of 2). After point A (as described in the previous figure), the API checks if the error queue is empty. If it is, the error state gets set to off. The API now compares the call's most severe error (denoted as C) and the application-defined severity (defined as A). If A is less than C, then the API returns to the application program. If A is greater than or equal to C, then an application-defined second-level error handler is invoked and the exit routine is called. If the error queue is empty, then the error state is set to on. The API checks if there is an application error handler. If there is, a first level error handler is invoked and the severity of the error is set. If there is not, the system logs the error and sets the severity of this error to zero. In both cases, the error severity is compared with the call's most severe error. The highest error is stored and the API goes back to Point A.

Error Detection

During the processing of a subroutine call, there are two categories of errors: (1) validation errors, and (2) processing errors.

Validation Errors

Before processing any subroutine call, the graPHIGS API shell checks the validity of the subroutine call based on the current state of the graPHIGS API system. For example, when a first-level error handler is given control the graPHIGS API system is put into an error state which prohibits most non-inquiry subroutines from being called. Therefore, if you write a first-level error handler that calls a non-inquiry subroutine such as Open Structure **(GPOPST)**, the graPHIGS API.

Validating a subroutine call includes checking the validity of the parameters supplied by your application. For example, if your application calls the Set Polyline Color Index **(GPPLCI)** subroutine with a color index which is less than zero, the graPHIGS API shell will generate an error because color index values must be greater than or equal to zero.

Processing Errors

After a subroutine call to the graPHIGS API is validated, the information generated by the subroutine call may be further processed by a graPHIGS API nucleus. Due to the buffering of data between a shell and nucleus, the actual processing of the information generated by a subroutine call to the graPHIGS API may occur at any time before or after the graPHIGS API shell has returned control to your application. While attempting to process the information, the nucleus may detect errors which are sent back to the graPHIGS API shell to be processed after the next subroutine call to the graPHIGS API.

Error Handling Mode and the Error Queue

During the processing of a graPHIGS API subroutine call, all errors detected by the graPHIGS API shell and nucleus are put onto the shell's error queue to be processed before control is returned to your application. Asynchronous errors generated by the graPHIGS API nucleus are also put on the shell's error queue but will not be processed until the next API call has been completed. The graPHIGS API system can detect and queue multiple errors, each of which are processed in the order they are queued.

In order to prevent the graPHIGS API from queueing errors, your application can turn off the API's error handling mode using the Set Error Handling Mode **(GPEMO)** subroutine. With the error handling mode set to `OFF`, errors are still detected and processed internally but are not queued and therefore not reported in any way. By default, the error handling mode is set to `ON` meaning that all errors are put on the error queue to be processed.

To boost the performance of a stable application, you can disable some of the parameter checking performed by the graPHIGS API shell. Parameter checking is disabled through the External Defaults File (EDF) or the Application Defaults Interface Block (ADIB) parameter of the Open graPHIGS **(GPOPPH)** subroutine. The use of EDF and ADIB options are discussed in detail in *The graPHIGS Programming Interface: Technical Reference*. When parameter checking is disabled, the graPHIGS API shell will only check the validity of the parameters necessary to build the data stream between the shell and the nucleus. The data stream itself will not be validated and could potentially cause the nucleus to abend if the data stream does contain an error. Therefore, turning off parameter checking will boost the performance of your application but also increases the risk that your application will cause an abend.

Error Queue Processing

The graPHIGS API does not explicitly notify your application of errors by way of parameters or return codes. Instead, after processing a subroutine call to the graPHIGS API, the shell will check the error

queue. If the queue is empty, the shell returns control to your application. If the queue is not empty, the shell invokes the error processing facility of the graPHIGS API By default, the error processing facility will log each error in the error file specified by the first parameter of the Open graPHIGS **(GPOPPH)** subroutine. For example, if your application calls the Set Interior Color Direct **(GPICD)** subroutine with a color parameter which is invalid, the following error will be logged in the error file:

```
GPICD AFM0096 COLOR PARAMETER OUT OF RANGE FOR CURRENT COLOR MODEL
```

Your application can change the way errors are processed by specifying first- and second-level error handlers as discussed in the following sections.

First-Level Error Handling

To override the default handling of errors your application must define a first-level error handler using the Define Error Handler **(GPEHND)** subroutine. **GPEHND** takes one parameter which is the address of your application's first-level error handler. To restore default error handling, your application must call the **GPEHND** subroutine with a parameter of zero.

With a first-level error handler defined, the graPHIGS API will perform the following actions for each error on the error queue:

1. Set the graPHIGS API system error state to `ON`
2. Invoke your application's first-level error handler, which is passed the following parameters:
   - a unique error identification number
   - the name of the graPHIGS API subroutine which detected the error
   - the name of the error log file specified when the graPHIGS API was opened
   - a flag that specifies whether the error was generated by a graPHIGS API subroutine that takes a workstation identifier (*wsid*) as a parameter
   - if the above flag is set, the *wsid* is also passed. Otherwise, the *wsid* parameter should be ignored
3. returns to 2 if more errors are on the queue
4. sets the graPHIGS API system error state to `OFF`.

To determine the cause of an error, your application's first-level error handler may call any inquiry subroutine to obtain information from state lists and description tables. For example, the error message text associated with the error being processed may be obtained through the Inquire Error Message **(GPQEMS)** subroutine. Inquiry subroutines do not generate errors; rather, they return error indicators describing the success or failure of the inquiry.

A first-level error handler may log the current error and its associated message text using the Error Logging **(GPELOG)** subroutine. **GPELOG** lets your application specify the name of the file into which the error is logged. Note that **GPELOG** is the only valid non-inquiry subroutine that can be called by the first-level error handler. Any other non-inquiry subroutine called by a first-level error handler will cause the error handler to be removed (undefined) resulting in the default processing of subsequent errors.

The major purpose of a first-level error handler is to assign a severity value between 0 and 16. The severity value controls whether the API calls the second-level error handler which has more freedom to modify the state of the graPHIGS API system as discussed in the following section.

Second-Level Error Handling

The graPHIGS API provides a default second-level error handler which displays an informational message only. You may choose to have your application provide a more sophisticated second-level error handler which can call non-inquiry subroutines unlike a first-level error handler.

For your second-level error handler to be called, it must have been previously defined by a call to the Specify Error Exit and Error Threshold **(GPEXIT)** subroutine and your first-level error handler must have set an error severity value greater than or equal to the severity threshold set using **GPEXIT** Calls that

complete normally are assigned a severity of zero by the API. If you want your second-level error handler to receive control after every API call then you must set the severity threshold level to zero.

Using a second-level error handler which calls the Synchronize **(GPSYNC)** subroutine can be helpful while debugging a program. **GPSYNC** forces the contents of the shell'communication buffer to be sent to the nucleus to be processes and will optionally wait for the nucleus to finish processing the buffer before returning control to your application. By waiting for the nucleus to finish processing, all errors contained in the buffered data will be detected by the nucleus and put on the shell's error queue before **GPSYNC** returns control to your application. With the severity threshold set to zero, your second-level error handler can call **GPSYNC** to force the contents of the shell to nucleus buffer to be sent and processed before your application receives control to make the next call to the graPHIGS API This technique is recommended for debugging purposes, only because it will greatly degrade the performance of your application.

Part 3. Appendixes

Appendix A. House Sample Program

The following pages contain a listing of the completed house sample program.

```
* DECLARE VARIABLES
      INTEGERx4 STATUS,CHOICE
      INTEGERx4 WSID,STRID(5),VIEW1
      REALx4 WINDOW(4),VIEWPT(4)
      REALx4 HOUSE(12)
      INTEGERx4 CHARID(2),VALUE(2)
      REALx4 MATRIX(9)
      INTEGERx4 POST
      REALx4 DOOR(10),TRANSD(9)
      INTEGERx4 ATLIST(3),ATFLAG(3)
      INTEGERx4 COLOR(2)
      REALx4 CTABLE(3)
      REALx4 TRANSW(9), HWNDW1(10),HWNDW2(4),HWNDW3(4)
      REALx4 WNDOW2(4),VIEWPA(4)
      INTEGERx4 VIEW2
      REALx4 AREA(6),CSIZE(3)
      INTEGERx4 DATAL,DATA(7),ERRIND,UNITS,ASIZE(3)
      CHARACTERx8 ACONID,AWSTYP
      INTEGERx4 IWSID,CLASS,DEVICE,ILEN,OLEN
      INTEGERx4 VDATAL,VDATA(4)
      REALx4 VVALUE,LOW,HIGH,SCALE(2),VMATRX(9)
      INTEGERx4 ACTNUM,ACTLEN,CDATA(2),TERM
* INITIALIZE VARIABLES
      DATA WSID  /1/
      DATA STRID /1,2,3,4,5/
      DATA VIEW1 /1/
      DATA WINDOW /-100.0,100.0,-100.0,100.0/
      DATA VIEWPT /0.0,0.5,0.0,0.5/
      DATA CHARID /6,8/
      DATA VALUE /2,2/
      DATA VIEW2 /2/
      DATA WNDOW2 /-50.0,50.0,-50.0,50.0/
      DATA VIEWPA /0.5,1.0,0.5,1.0/
      DATA HOUSE/0.0,0.0,0.0,40.0,30.0,70.0,
     *           60.0,40.0,60.0,0.0,0.0,0.0/
      DATA MATRIX /1.0,0.0,0.0,
     *             0.0,1.0,0.0,
     *            -30.0,-35.0,1.0/
      DATA DOOR /0.0,0.0,0.0,20.0,10.0,20.0,
     *           10.0,0.0,0.0,0.0/
      DATA TRANSD /1.0,0.0,0.0,
     *             0.0,1.0,0.0,
     *            10.0,0.0,1.0/
      DATA POST /2/
      DATA ATLIST /1,2,3/
      DATA ATFLAG /2,1,2/
      DATA COLOR /1,2/
      DATA CTABLE /0.40,0.25,0.25/
      DATA HWNDW1 /0.0,0.0,0.0,10.0,16.0,10.0,16.0,0.0,0.0,0.0/
      DATA HWNDW2 /0.0,5.0,16.0,5.0/
      DATA HWNDW3 /8.0,0.0,8.0,10.0/
      DATA TRANSW / 1.0, 0.0,0.0,
     *              0.0, 1.0,0.0,
     *             37.0,25.0,1.0/
      DATA DATAL /28/
      DATA DATA /4,0,0,2,1,1,2/
      DATA ILEN /8/
      DATA VDATAL /16/
      DATA VDATA /1,0,0,10/
      DATA LOW /0.5/
      DATA HIGH /2.0/
* OPEN FUNCTIONS
```

```
      CALL GPOPPH('SYSPRINT',0)
      CALL GPCRWS(WSID,1,7,'IBM5080','5080    ',0)
* VIEW DEFINITION
      CALL GPXVR(WSID,VIEW1,14,VIEWPT)
      CALL GPXVR(WSID,VIEW1,16,WINDOW)
      CALL GPXVCH(WSID,VIEW1,1,8,2)
      CALL GPXVR(WSID,VIEW2,14,VIEWPA)
      CALL GPXVR(WSID,VIEW2,16,WNDOW2)
      CALL GPXVCH(WSID,VIEW2,2,CHARID,VALUE)
* DATA CREATION
      CALL GPCR(WSID,6,1,CTABLE)
      CALL GPXPLR(WSID,1,1,1)
      CALL GPXPLR(WSID,1,2,2.0)
      CALL GPXPLR(WSID,1,3,COLOR)
*
      CALL GPOPST(STRID(1))
      CALL GPMLX2(MATRIX,POST)
      CALL GPASF(3,ATLIST,ATFLAG)
      CALL GPPLI(1)
      CALL GPPLCI(5)
      CALL GPPL2(6,2,HOUSE)
      CALL GPEXST(STRID(2))
      CALL GPEXST(STRID(3))
      CALL GPCLST
*
      CALL GPOPST(STRID(2))
      CALL GPMLX2(TRANSD,POST)
      CALL GPINLB(1)
      CALL GPLT(6)
      CALL GPPLCI(6)
      CALL GPPL2(5,2,DOOR)
      CALL GPCLST
*
      CALL GPOPST(STRID(3))
      CALL GPMLX2(TRANSW,POST)
      CALL GPPLCI(4)
      CALL GPPL2(5,2,HWNDW1)
      CALL GPPL2(2,2,HWNDW2)
      CALL GPPL2(2,2,HWNDW3)
      CALL GPCLST
* DATA DISPLAY
      CALL GPASSW(WSID,1)
      CALL GPARV(WSID,VIEW1,STRID(1),1.0)
      CALL GPARV(WSID,VIEW2,STRID(1),1.0)
      CALL GPUPWS(WSID,2)
* INPUT FUNCTIONS
      CALL GPQRCT(WSID,ILEN,ERRIND,OLEN,ACONID,AWSTYP)
      CALL GPQDS(AWSTYP,ERRIND,UNITS,CSIZE,ASIZE)
      IF(ERRIND.NE.0) GOTO 200
      AREA(1) = 0.
      AREA(2) = CSIZE(1)
      AREA(3) = 0.
      AREA(4) = CSIZE(2)
      AREA(5) = 0.
      AREA(6) = CSIZE(3)
      CALL GPINCH(WSID,1,1,2,AREA,DATAL,DATA)
      CALL GPCHMO(WSID,1,3,2)
      CALL GPINVL(WSID,1,1.,4,AREA,LOW,HIGH,VDATAL,VDATA)
      CALL GPVLMO(WSID,1,3,2)
      TIME = 10.
 100  CALL GPAWEV(TIME,IWSID,CLASS,DEVICE)
      IF(CLASS.EQ.4) THEN
        IF(DEVICE.EQ.1) THEN
          CALL GPGTCH(CHOICE)
          IF(CHOICE.EQ.1) GOTO 200
          IF(CHOICE.EQ.4) THEN
            CALL GPOPST(STRID(2))
```

```
            CALL GPEDMO(2)
            CALL GPEPLB(1)
            CALL GPOEP(1)
            CALL GPQED(1,32,ERRIND,ACTNUM,ACTLEN,CDATA,TERM)
            IF(CDATA(2).EQ.6) CALL GPLT(1)
            IF(CDATA(2).EQ.1) CALL GPLT(6)
            CALL GPCLST
            CALL GPUPWS(WSID,2)
          ENDIF
        ENDIF
      ENDIF
      IF(CLASS.EQ.3) THEN
        IF(DEVICE.EQ.1) THEN
          CALL GPGTVL(VVALUE)
          SCALE(1) = VVALUE
          SCALE(2) = VVALUE
          CALL GPSC2(SCALE,VMATRX)
          CALL GPXVR(WSID,VIEW1,18,VMATRX)
          CALL GPUPWS(WSID,2)
        ENDIF
      ENDIF
      GOTO 100
* CLOSE FUNCTIONS
 200  CALL GPCLWS(WSID)
      CALL GPCLPH
      STOP
      END
```

Appendix B. Compatibility

Compatibility with Version 1 of the graPHIGS API

As new functions have been added to the graPHIGS API products, every effort has been made to make changes transparent to existing applications. The objective has been for a Version 1 application to run in a Version 2 environment, using Version 2 defaults. If you look at *The graPHIGS Programming Interface: Subroutine Reference*, you will notice that there are a number of subroutines with overlapping functions. An example of this is the Open Workstation **(GPOPWS)** and Create Workstation **(GPCRWS)** subroutines. The **GPCRWS** subroutine is new for Version 2, and adds the parameters needed for the advanced functionality. On the other hand, the older subroutine, **GPOPWS**, continues to work with a set of assumed defaults. This lets older applications work in the new environment, while new functions and capabilities can be added at your discretion.

Compatibility Subroutines

The following table lists all of the graPHIGS API Version 1 subroutines which are now considered "compatibility" subroutines, and Version 2 subroutines that should be used instead.

Table 5. Subroutines

Compatibility Subroutines	Version 1	Version 2
Inquire Actual Break Capabilities	GPQABK	GPQBK[1]
Inquire Actual Color Facilities	GPQACF	GPQCF[1]
Inquire Actual Maximum Display Surface Size	GPQADS	GPQDS[1]
Inquire Actual Edge Facilities	GPQAEF	GPQEF[1]
Inquire Actual Available Escape Functions	GPQAES	GPQES[1]
Inquire Actual Font Pool Size	GPQAFP	GPQFP[1]
Inquire List of Actual GDPs	GPQAGD	GPQGDP[1]
Inquire Actual Interior Facilities	GPQAIF	GPQIF[1]
Inquire Actual Input Character Set Facilities	GPQAIS	GPQISF[1]
Inquire Actual Trigger Capabilities	GPQAIT	GPQIT[1]
Inquire Actual Polyline Facilities	GPQALF	GQPLF[1]
Inquire List of Actual Logical Input Devices	GPQALI	GPQLI[1]
Inquire Actual Length of Workstation Tables	GPQALW	GPQLW[1]
Inquire Actual Polymarker Facilities	GPQAMF	GPQPMF[1]
Inquire Actual Number of Definable Views	GPQANV	GPQNV[1]
Inquire Actual Pattern Facilities	GPQAPF	GPQPAF[1]
Inquire Actual Primary Character Set	GPQAPS	GPQPCS
Inquire Actual View Facilities	GPQAVF	GPQVF[1]
Inquire Actual Workstation Category	GPQAWC	GPQWC[1]
Inquire Actual Workstation Display Class	GPQAWD	GPQWD[1]
Inquire Color Representation	GPQCR	GPQXCR
Inquire Current Viewing Transformation	GPQCVX	GPQCVR
Inquire Edge Representation	GPQER	GPQXER[1]
Inquire Element Content	GPQE	GPQED[4]
Inquire Element Type and Size	GPQETS[5]	GPQEHD

Table 5. Subroutines (continued)

Compatibility Subroutines	Version 1	Version 2
Inquire Interior Representation	GPQIR	GPQXIR[1]
Inquire Polyline Representation	GPQLR	GPQXLR[1]
Inquire Polymarker Representation	GPQMR	GPQXMR[1]
Inquire Requested Viewing Transformation	GPQRVX	GPQCVR
Inquire Text Representation	GPQTR	GPQPTR[1]
Inquire Text Facilities	GPQTXF	GPQXTX[1]
Inquire Workstation Connection and Type	GPQWCT	GPQRCT
Open Workstation	GPOPWS	GPCRW [3]
Set View Characteristics	GPVCH	GPXVR
Set View Mapping 2	GPVMP2	GPXVR[2]
Set View Mapping 3	GPVMP3	GPXVR[2]
Set View Matrix 2	GPVMT2	GPXVR[2]
Set View Matrix 3	GPVMT3	GPXVR[2]
Set Color Representation	GPCR	GPXCR
Set Edge Representation	GPER	GPXER
Set Interior Representation	GPIR	GPXIR
Set Polyline Representation	GPPLR	GPXPLR
Set Polymarker Rep	GPPMR	GPXPMR
Set Text Rep	GPTXR	GPXTXR

Notes

1. Actual WDT

Actual workstation characteristics are now contained in the actual Workstation Description Table (WDT), instead of the Workstation State List (WSL), although WSL inquiries continue to work for compatibility. This change was made to be consistent with the PHIGS standard. Inquiries directed at the generic WDT are (as before), likely to be inaccurate. To make inquiries to the actual WDT, the application must first obtain the “realized” workstation type based on the workstation’s identifier (*wsid*) using the Inquire Realized Connection and Type **(GPQRCT)** subroutine. The workstation type returned by **GPQRCT** can then be used in any WDT inquiry to obtain the actual capabilities of the workstation.

2. GPXVR

Supersedes existing view characteristics subroutines **(GPXVCH)** and **(GPVCH)**, and adds those characteristics required to support viewing extensions such as HLHSR mode.

3. GPCRWS

The Create Workstation **(GPCRWS)** subroutine supercedes the Open Workstation **(GPOPWS)** subroutine. It adds additional parameters needed to specify resources other than those taken by default with **GPOPWS** For example, in compatibility mode, on **GPOPWS**, a nucleus of 1 is assumed, yet if using **GPCRWS** you can specify the nucleus id.

4. GPQED

The Inquire Element Data **(GPQED)** subroutine returns the structure element information that currently resides in Version 2 structure store. The Inquire Element Type and Size **(GPQETS)** and Inquire Element Content **(GPQE)** subroutines still work, and return element data in the Version 1 format, but work only for Version 1 structure elements.

The output from the Inquire Structure Element **(GPQE)** subroutine is not always exactly the same as in Version 1. For example, for Pixel structure elements, the Pixel Packing Factor is always 1 (8 bits/pixel).

5. GPQETS:

The element size returned by Inquire Element Type and Size **(GPQETS)** is that of the Version 2 element. In some cases, this is not the same size as in Version 1, and it may not be the size returned by the Inquire Element Content **(GPQE)** subroutine. For example, Insert Application Data **(GPINAD)** structure element lengths are rounded to a multiple of 4 bytes. This size is different from that returned in Version 1, and is different from the value of the number of bytes parameter returned when the Inquire Element Content compatibility subroutine is called for the same structure element.

Application Control of Compatibility Using ADIB/EDF

Existing applications can run with no changes by using defaults supplied by the graPHIGS API The following options have been defined to override the defaults:

COMBSZ	Size of I/O Buffers (Default 64K).
DEFACTF	If compatibility is requested (default) fonts are sent to the workstation, similar to Version 1.
DEFNUC	Default = Call Communication method. This is essentially the same as Version 1.
IQSIZE	Event queue size in bytes (default is 16K).
HCHECK	Shell syntax (parameter list) checking (default = yes).

Color Compatibility

Applications written to the Version 1 interface will continue to display color correctly. However, if you wish to use direct color, set the appropriate value in the color tables to ensure that reasonable colors are displayed. See Chapter 17. Manipulating Color and Frame Buffers for more information.

Input Compatibility

Several input changes have been introduced into the graPHIGS API which may effect current Version 1 applications.

Data records used to initialize input devices are checked for errors more thoroughly in Version 2 than in Version 1. For example, in Version 1 your application could initialize a choice device which has only 12 buttons with a data recorded assuming the device has 32 buttons. Since the data record was not checked against the actual number of available buttons, no error was generated. In version 2 an error is generated and the data record is ignored.

In Version 1 of the graPHIGS API, the buffer length parameter of the Initialize String **(GPINST)** subroutine included the length of the prompt string. In Version 2 the buffer length parameter is defined to be independent of the prompt string length.

In Version 1 of the graPHIGS API, the Set Device Mode subroutines were sent immediately to the workstation. In Version 2, these commands are buffered until an event causes the buffer to be sent to the nucleus. In most cases, this change should not be a problem as the buffers are sent when the application uses a subroutine to obtain input device data. Any of the following graPHIGS API subroutine calls will cause the buffer to be sent to the workstation:

- REQUEST xxx DEVICE (**GPRQxx**)
- SAMPLE xxx DEVICE (**GPSMxx**)
- AWAIT EVENT **(GPAWEV)** when the event queue is empty

The buffer can also be forced to the nucleus by use of the Synchronixe **(GPSYNC)** subroutine.

Error Messages

New message numbers have been assigned for Version 2. Note that all error numbers are the same for compatibility. The new message numbers are in the following ranges:

- 1300 - 1399 DAP Messages
- 2000 - 2100 Device Driver Messages.

Error messages for asynchronous errors now contain asterisks in the API subroutine name field. In addition, some subroutines generate the same shell-to-nucleus datastream (Open Workstation and Create Workstation for example). An error message may contain a ″*″ after the API subroutine name. The ″*″ indicates that the actual name of the subroutine which caused the error could not be determined. See *The graPHIGS Programming Interface: Messages and Codes* for a detailed explanation of error messages.

Workstation Compatibility

The Message **(GPMSG)** subroutine text is not sent immediately to the workstation. If you require the **GPMSG** text to be displayed immediately, use the Synchronize **(GPSYNC)** to force the message text be sent.

In Version 1, a workstation in ASAP deferral mode received and processed each subroutine call immediately. In Version 2, a workstation's deferral mode is affected by the buffering done by the shell. Subroutine calls are buffered until sent by the shell, and once received by the workstation, they are processed according to the deferral mode. This buffering may impact the operation of your application.

In Version 1, unsupported class names defaulted to the highest available class name on the workstation. In Version 2, unsupported class names are ignored.

Structure Element Compatibility

(GPTXPT) subroutine applies to both geometric and annotation text. Version 2 provides the Set Annotation Path **(GPAPT)** attribute, which sets the path for annotation text. The Set Text Path attribute applies only to geometric text in Version 2.

Error Handling

Although the error handling mechanism of Version 1 of and Version 2 of the graPHIGS API is the same, the timing of when errors are reported to your application may differ. The timing difference is due to the buffering of data between the shell and the nucleus of the Version 2 product. In Version 2 some errors are not detected by the shell but are detected when a request is actually processed by a graPHIGS API nucleus. Because the request may be kept in a buffer for some period of time, predicting when a request will be processed is generally impossible. An error detected by a nucleus is reported to the shell asynchronously and queued on the shell's error queue. The shell will report the error before returning control to your application on the current or next call to the API.

The following is a list of subroutines and related errors which can only be detected by a graPHIGS API nucleus and therefore may not be generated during the call which caused the error.

- Initialize Input Device (**GPINxx**)

  Error 141: INPUT DEVICE NOT IN CORRECT MODE

  The validity of these functions depends on the current operating mode of the device which may be changed by other application processes sharing a workstation.
- Set Input Device Mode (**GPxxMO**)

  Error 168: INPUT DEVICE IS CURRENTLY OWNED BY ANOTHER APPLICATION

  If the device was activated and therefore currently owned by another application process, the application process which received error 168 cannot change the device's operating mode.

- Delete Element Between Labels **(GPDELB)**

  Error 130: LABEL IDENTIFIER CANNOT BE FOUND IN THE OPEN STRUCTURE

  Because structure contents exist only within a nucleus, a shell cannot check for the existence of labels.
- Delete Element Group **(GPDLEG)**

  Error 130: LABEL IDENTIFIER CANNOT BE FOUND IN THE OPEN STRUCTURE

  Because structure contents exist only within a nucleus, a shell cannot check for the existence of labels.
- Set Element Pointer at Label **(GPEPLB)**

  Error 130: LABEL IDENTIFIER CANNOT BE FOUND IN THE OPEN STRUCTURE

  Because structure contents exist only within a nucleus, a shell cannot check for the existence of labels.
- Set Element Pointer at Pick Identifier **(GPEPPK)**

  Error 566: PICK IDENTIFIER CANNOT BE FOUND IN THE OPEN STRUCTURE.

  Because structure contents exist only within a nucleus, a shell cannot check for the existence of pick identifiers.
- Copy Element Group **(GPCPER)**

  Error 125: ATTEMPTING TO EXECUTE THE OPEN STRUCTURE

  Because structure contents exist only within a nucleus, a shell cannot check for the existence of an execute structure-type element that references the open structure in the elements to be copied.
- Copy Structure **(GPCPST)**

  Error 125: ATTEMPTING TO EXECUTE THE OPEN STRUCTURE

  Because structure contents exist only within a nucleus, a shell cannot check for the existence of an execute structure-type element that references the open structure in the elements to be copied.
- Change Structure Identifier **(GPCSI)**

  Error 129: ATTEMPTING TO HAVE THE RESULTING STRUCTURE EXECUTE ITSELF

  Because structure contents exist only within a nucleus, a shell cannot check for the existence of an execute structure-type element in the original structure that references the resulting structure.
- Change Structure References **(GPCSRS)**

  Error 129: ATTEMPTING TO HAVE THE RESULTING STRUCTURE EXECUTE ITSELF

  Because structure contents exist only within a nucleus, a shell cannot check for the existence of an execute structure-type element in the resulting structure that references the original structure.
- Change Structure Identifier and References **(GPCSIR)**

  Error 129: ATTEMPTING TO HAVE THE RESULTING STRUCTURE EXECUTE ITSELF

  Because structure contents exist only within a nucleus, a shell cannot check for the existence of an execute structure-type element in the original structure that references the resulting structure.
- Set Element Pointer at Code **(GPEPCD)**

  Error 132: ELEMENT CODE DOES NOT EXIST BEFORE END OF STRUCTURE

  Because structure contents exist only within a nucleus, a shell cannot check for the existence of an element with the specified code.
- Any workstation related subroutines.

  Workstation related errors have error numbers larger than 900.

  Because these errors are detected during processing within a nucleus, they may occur at any time after a request is sent to the nucleus.

The possibility of the asynchronous errors occurring for the Set Input Device Mode (**GPxxMO**) subroutines could cause problems for you when using the Await Event **(GPAWEV)** subroutine. For example, if your application calls the Set Choice Mode **(GPCHMO)** subroutine to put a choice device in EVENT mode and the device is already in use by another application process, the request will fail and the nucleus will send an asynchronous error back to the shell. But, since the Set Choice Mode request is buffered and not sent to the nucleus until the **GPAWEV** subroutine is called, your application will not know that the Set Choice

Mode request failed and the **GPAWEV** subroutine would wait until a timeout occurs. A shell cannot detect this situation at the time the **GPCHMO** subroutine is called but will be informed via an asynchronous error when it is waiting for the actual device event. Therefore, the graPHIGS shell resolves the outstanding wait condition of the **GPAWEV** subroutine when it receives asynchronous errors. In this case, the event class returned by **GPAWEV** will be 0=NONE which it does not necessarily mean that the timeout interval has expired, but does mean that the event queue is empty.

Delayed error reports for structure editing subroutines like the Set Element Pointer at Label **(GPEPLB)** subroutine may cause more serious problems for applications written to Version 1 of the graPHIGS API Consider a case where the application wants to change the colors of all polylines in a structure. Assuming that each polyline is preceded by a label element with sequential numbers and a set polyline color index element, the following calling sequence could then be used to change all of the polyline color indexes elements:

```
i = the_first_number
Set Element Pointer at Label (i)
do while no error
  Offset Element Pointer (1)
  Delete Element
  Set Polyline Color Index (new_color)
  i = i + 1
  Set Element Pointer at Label (i)
end
```

This sequence works in the Version 1 graPHIGS API, but in the Version 2 environment, it will cause a problem. Because the error ″130: LABEL IDENTIFIER CANNOT BE FOUND IN THE OPEN STRUCTURE″ will be reported asynchronously, there is no assurance that the loop terminates at the correct place. Before the error is reported to the shell, some delete requests have been already generated in the shell's I/O buffer and will be eventually sent to the nucleus to be executed, resulting in more errors.

To solve this problem a synchronization mode is introduced as a parameter of the Synchronize **(GPSYNC)** subroutine. The synchronization mode has one of the following two values and specifies how the communication between the shell and nucleus should be performed when the **GPSYNC** subroutine is called.

1=NOWAIT
: No synchronization is performed. Each time **GPSYNC** is called, any buffered data is sent to the nucleus and control is immediately returned to the application. The nucleus does not generate any synchronization information.

2=SYNC_WAIT
: Synchronization is performed. Each time **GPSYNC** is invoked, any buffered data is sent to the nucleus and the shell enters a wait state until the nucleus responds that all outstanding requests from that connection have been processed.

The *wait* parameter ensures that all requests created by previous graPHIGS API subroutine calls have been sent to the target nucleus and their processing has been completed before control is returned to your application. Therefore, for the example case, the following calling sequence ensures the correct result:

```
i = the_first_number
Set Element Pointer at Label (i)
Synchronize (WAIT)
do while no error
  Offset Element Pointer (1)
  Delete Element
  Set Polyline Color Index (new_color)
  i = i + 1
  Set Element Pointer at Label (i)
  Synchronize (WAIT)
end
```

The above pseudo code will work but is not recommended for performance critical applications.

Appendix C. graPHIGS Glossary

4 x 4 matrix. A logical device used in all transformations to modify the coordinates of a primitive.

A

address space. The set of addressable points, often in a specific coordinate direction; usually limited by the data path width in a physical device. Some addressable points may not be visible if they fall outside the display surface boundaries.

addressable point. Any point in the device coordinate space, whether displayable or not.

ambient light. A type of light source that enters into the reflectance calculation independent of the area being illuminated or the location of the eye.

ambient reflection coefficient. The fraction of light from ambient light sources that are reflected from an area.

anisotropic mapping. A transformation which need not preserve aspect ratio.

annotation text. An output primitive for text display for which only the text position is transformed. This is used for annotating drawing when the application wants the text always facing the display screen operator. The graPHIGS API uses hardware character generators where possible when generating annotation text output.

antialiasing. A technique used to ″smooth″ the jagged-edged appearance of graphical elements (e.g., lines, polygons) caused by scan conversion into component pixels on a raster grid. Techniques may include varying the intensity of pixels near the ″jagged″ primitive or the pixels associated with the primitive itself.

application data. Non-graphical data used by an application program. Application specific data may be inserted into a structure in the form of the application data structure element.

application message. A message passing facility provided by the graPHIGS API to allow application processes to communicate.

application program. A program used to generate graphics output and/or receive graphics input.

area defining primitive. Any primitive whose attributes may be derived from the interior representation table. Fill area, fill area set, triangle strip, and non-uniform B-spline surface are considered to be area defining primitives.

area properties. An attribute that defines the set of reflectance coefficients and other aspects which provide information about an area defining primitive to the rendering pipeline. Area properties include ambient reflection coefficient, diffuse reflection coefficient, specular reflection coefficient, specular exponent and transparency coefficient.

aspect ratio. The ratio of an object's width to its height. In graphics it usually pertains either to the display surface, to a window or view port, or to the shape of characters.

attenuation coefficient. A parameter that models the fading of light over a distance from one point to another. This phenomenon is referred to as the attenuation of light.

attribute source flag (ASF). A flag indicating whether a primitive's attribute is selected from a particular workstation-dependent attribute bundle table, or by way of an individual attribute specification. By default, ASFs are set for individual specification.

attribute. A particular property that applies to an output primitive. Attribute values are modal.

attribute binding. The application of attributes to the output primitives that they affect. In the graPHIGS API, this occurs at structure traversal time, not when output primitive elements are inserted into a structure.

attribute bundling. See Bundle, Bundle Table.

attribute definition. The process of declaring the set of possible values for an attribute. Attribute definition for workstation-dependent attributes is achieved through definition of the contents of the workstation attribute tables in the Set Extended Representation functions. Attribute definition functions are control functions, not structure elements.

attribute index. Some attributes are organized into workstation-dependent tables, which are accessed by way of workstation-independent attribute indexes.

attribute selection. The means by which the user specifies attribute values from a set of possible alternatives. For workstation attribute tables, the possible alternatives are the table entries. Attribute selection functions are structure elements. See also ASF.

attribute specification. The processes of attribute definition and attribute selection.

attributes of primitives. Ways in which the characteristics of an output primitive can vary. Some aspects are controlled directly by workstation-independent attributes; some can be controlled indirectly through a bundle table. Transformation and viewing matrices, as well as highlighting and visibility, are treated as attributes in the graPHIGS API.

axis. One of the reference lines of a coordinate system.

B

back face. Any portion of an area defining primitive which has a negative scalar product of its geometric normal as the vector connecting the positive area to the eyepoint. Back facing portions of primitives can be subjected to special attributes or made invisible.

binding. See Attribute Binding, Language Binding.

break action. An implementation-dependent and workstation-dependent means for the operator to interrupt an input operation.

bundle. A group of related attributes which may be selected and modified as a unit.

bundle attribute selection. Selection of all attributes applying to an output primitive by selecting a single bundle table index.

bundle table. A workstation-dependent table associated with particular output primitives. Each entry in a bundle table contains information specifying all the workstation-dependent aspects of a primitive. In the graPHIGS API, bundle tables exist for polylines, polymarkers, texts, interiors, and edges. The information in bundle tables is accessed via a bundle table index, one for each bundle table.

bundle table entry. A single entry in a bundle table. Each entry contains one value for each attribute aspect which applies to the corresponding output primitive. This set is workstation-dependent, meaning that a bundle table entry on different workstations may contain different values (although the format is always the same).

bundle table index. A number (value) used to access a particular bundle table entry. (Index is a *mechanism* for selection, attributes!) The index is workstation-independent. There is one bundle table index for each bundle table, e.g., TEXT INDEX for the text bundle table.

C

cartesian coordinates. Two-dimensional (X, Y) or three-dimensional (X, Y, Z) coordinates used in determining the position of a point in relation to a set of orthogonal axes.

center of projection. See Projection Reference Point (PRP).

character expansion factor. An attribute of text which specifies the multiplicative deviation of character aspect ratio from the defined nominal value. (A character expansion factor value of 1.0 specifies that the nominal character aspect ratio defined for the text font to be used).

character height. An attribute of text which specifies the height of a capital character in modeling coordinates (from Base Line to Cap Line).

character plane. The plane on which text appears. See Text Plane.

character set. Character set is an attribute of text. The text character set identifies the language of the text. Depending on the language, character codes may either be one or two bytes and their interpretation varies.

character spacing. An attribute of text which specifies the fraction of character height to be added between adjacent character boxes in a string. (A value of 0.0 specifies that no additional spacing be used.)

character up vector. An attribute of text which indicates the up direction of the text characters. The character up vector is a two-dimensional vector on the text plane.

child structure. A structure which is invoked by an ″execute structure″ element in another structure. See also parent structure.

choice. An input class which returns a positive integer representing an enumeration of alternatives.

choice device. (1) A logical input device that provides one value from a set of alternatives; a *choice* logical input device can be implemented as a physical input device such as the Lighted Program Function Keyboard (LPFK) or the function keys on a keyboard. (2) See also pick device, locator device, valuator device.

chordal deviation. Chordal deviation is the perpendicular distance between the actual curve and the approximating line segment.

class. An element of the class set; a classification of output primitives which is meaningful to application programs. A primitive will be detectable, highlighted, or invisible if one of the classes which are members of the set associated with the primitive is also contained in the inclusion filter for that attribute and none of the classes are in the exclusion filter.

class set. The traversal state which is comprised of classes which are added and removed during traversal. See Class.

clipping. Removing those parts of output primitives which lie outside a given boundary, usually a window, viewport, or view volume.

clipping volume. A volume in the viewing coordinate system. The clipping volume is defined by the window, near and far clipping planes, and projectors of the corners of the window. Graphics data within this volume will not be clipped. Data on the planes forming the edges for this volume are considered to be within the volume.

color approximation. See *color quantization*.

color index. An index used to access an entry of a color table in the graPHIGS API Each type of output primitive has its own color index, e.g., ″polyline color index.″

color mapping. See *color quantization*.

color mapping method. A method for quantizing colors.

color model. The method by which the user describes a color to the graphics system. The color model is a specification of a three-dimensional color coordinate system and a three-dimensional subspace in the coordinate system where each color is represented by a point. The RGB, CMY and HSV color models are supported by graPHIGS API.

color processing. See *color Quantization*.

color quantization. The process of mapping a arbitrary color vector to a color which can be displayed on a workstation.

color table. A workstation-dependent table, in which the entries specify the values defining a particular color.

color vector. Three values representing a three dimensional vector in normalized color space which define a specific color.

component frame buffer. A type of frame buffer consisting of three parts. Each part holds a display color table index for the red, green, and blue component of each screen pixel. Also see Frame Buffer and Indexed Frame Buffer.

composite primitives. Primitive which can be defined by intermixing different types of geometry such as lines, curves, surfaces, and arcs.

composite transformation. During structure traversal, the matrix resulting from the concatenation of the local modeling and the global modeling transformations.

concentration exponent. A measure of the concentration of light emanating from a spot light source relative to the direction of that source. This measure is used as an exponent to the cosine of the angle of deviation from the spot light direction, to produce a concentration scale factor for that deviation.

conceptual registers. storage device maintained at each workstation containing the current values of output primitive attributes; during display traversal, the API uses the current values in the registers to draw each primitive. See *workstation attribute registers*.

condition flags. An indexed state of boolean condition indicators that is part of the traversal state list. Some structure elements set or reset these indicators, other structure elements refer to the condition flags to determine their effects.

conditional traversal. The process of selectively traversing structures or portions of structures based on a condition determined at traversal time. Condition flags can be used to control traversal.

cone of influence. A cone that represents the influence of a spot light source. Only primitives (or portions of them) inside this cone are influenced by the lighting effect of the corresponding spot light source. The cone of influence is determined by the position, direction, and spread angle of the spot light source.

connection identifier. An implementation specific means of identifying a physical device, or a collection of these comprising a single workstation.

contour. A polygonal line that is closed and planar. Unlike a trimming loop, a contour may cross itself.

coordinate. A numerical value designating a position on an axis.

cull. To replace the representation of one or more primitives by another, typically similar, representation. The graPHIGS API defines two types of culling, face culling and extent culling.

D

DAP. See *Distributed Application Process*.

data storage. A conceptually centralized collection of structures. graPHIGS API structure editing and manipulation functions act on the contents of the data storage.

deferral mode. Part of the deferral state. The deferral mode specifies when changes to the workstation state must be reflected in the displayed image. Values of deferral mode are (ASAP, BNIL, BNIG, ASTI, WAIT).

deferral state. A control state which determines how long actions requested by the application program may be delayed.

default attribute value. The attribute value inherited by a root structure.

degenerate primitive. An output primitive whose definition reduces the dimensionality of the primitive. For example, if the vertices of a polygon are all co-linear, the polygon is degenerate.

depth cue. Colors of primitives are combined with a specific depth cue color to create an effect that distinguishes portions or primitives further from the viewer from those that are closer to the viewer.

depth cue mode. A field within a depth cue table entry that indicates whether or not depth cueing should be performed.

depth cue planes. Two reference planes in NPC parallel to the XY plane at which depth cue scaling factors are specified.

depth cue scale factors. Weights that determine how the primitive's color is combined with the depth cue color to produce the depth cue effect. One scale factor is specified for each depth cue plane.

depth cue table. A workstation dependent table, similar to the view table, which contains information necessary to control depth cueing.

depth queuing. The process of changing a primitive's color based on it's distance from the viewer to give the viewer a feeling of depth.

descendant. see child structure.

description table. A conceptually defined data structure containing static (fixed) information. Application programs can obtain description table contents through inquiry. See Description Table, Workstation Description Table.

device coordinates (DC). The coordinates used by workstation hardware. The coordinate units may be meters for devices such as a plotter, or an arbitrary sub-range of the integers for devices such as raster CRT's.

device space. The space defined by the addressable points of a display device.

diffuse reflection. An approximation of the light reflected equally in all directions from a surface. Diffusely reflected light gives a surface a dull matte appearance from all viewing angles.

diffuse reflection coefficient. The fraction of light from non-ambient light sources diffusely reflecting from a surface.

dimensionality. The number of coordinates needed to specify a point in any coordinate space. The graPHIGS API uses both two and three dimensional coordinate spaces.

direct color. A non-indexed method of specifying color where the components of the color are specified together with the color model in which these components are expressed.

directional light. A type of light source that enters into the reflectance calculation dependent on the area being illuminated, yet independent of the relative position of the area being illuminated.

disjoint polyline. An output primitive which allows a set of unconnected lines to be defined and manipulated one one structure element.

display color table. A display color table that typically corresponds to a workstation's hardware color table and contains the color vectors which are actually to be used to drive the display device such as a color monitor. A Display Color Table is sometimes referred to as an output color table because the output of the rendering pipeline is used to select colors from the display color table.

display device. A graphics device (for example, refresh display, storage tube display, plotter) on which pictures can be represented. A display device is one possible component of a workstation.

display space. That portion of the device space corresponding to the area available for displaying pictures. The display space may be all or part of the display surface.

display surface. The physical area on a display device onto which a graphics picture is generated (for example, the screen of a display or a computer printout). Also called ″view surface.″

distributed application process (DAP). Part of an application program which can run as a separate process either on the same node as the main application or on a separate node.

E

echo. The immediate notification of the current value, or measure of a logical input device, provided to the operator, usually at the display console.

echo type. A parameter of device initialization which selects the echoing technique for a particular logical input device.

edge flag. An indicator that is part of the specification for some output primitives that controls whether an individual edge is visible when the EDGE FLAG attribute is set to ON. If the EDGE FLAG attribute is set to OFF, edge visibility flags are not used.

editing. The modification of a structure, including its initial creation.

edit Mode. A mode controlling whether or not new structure elements are inserted between already existing structure (INSERT mode) elements or in place of them (REPLACE mode).

element pointer. A pointer used during structure editing which points to the structure element which is currently subject to modification.

element record. The inquirable contents of a structure element.

element type. A structure element's identifying classification, for example, POLYGON, LABEL, APPLICATION DATA, LINEWIDTH SCALE FACTOR, etc.

empty structure. A structure containing no structure elements.

error condition. An abnormal situation detectable by the graphics system.

error detection. The realization of the presence of an error condition.

error logging. Recording information of the occurrence of an error condition.

error reaction. A predefined action taken after error detection.

error reporting. The communication of the error condition to the application program.

escape. A function used to access implementation or device dependent features, other than for the generation of graphical output, that are not otherwise addressed by the GKS or PHIGS models.

event input. One of three input modes. In event input mode, asynchronous input is placed on the event queue when a trigger fires.

event queue. A time-ordered collection of input events.

event Report. A logical input value saved on an event queue.

exclusion highlighting set. Those highlighting classes whose primitives are declared ineligible for highlighting.

exclusion invisibility set. Those invisibility classes whose primitives are declared ineligible for invisibility.

exclusion pick set. Those pick classes whose primitives are declared ineligible for picking.

extended pixel memory (EPM). A feature available on an IBM 6090 which provides a 6090 with Component Frame Buffer capabilities.

eye point. See *projection reference point*.

F

face culling. The process of making back facing (or front facing) primitives or portions of primitives invisible.

facet. A planar portion of an area defining primitive. Facets may be defined by vertices that constitute a subset of the primitive's set of vertices. The following primitives consist of one or more facets: fill area, fill area set, and triangle strip. Facets may also result from the tessellation of parametric surface primitives.

facet normal. A vector supplied with an area defining primitive and which is associated with a facet of such a primitive. A facet normal is typically used for determining the geometric normal of a facet, and for determining the reflectance normal, in case vertex normals are not supplied with the primitive.

far clipping plane. A plane parallel to the view plane whose location is specified by a viewing coordinate space distance along the view plane normal from the view reference point. When far plane clipping is enabled, portions of objects behind the far clipping plane are discarded. Clipping against the far clipping plane is controlled separately from all other clipping.

far plane clipping. Clipping against the far clipping plane. If far plane clipping is enabled, only that part of the object which lies in front of and on the far clipping plane will be projected onto the viewport. Far plane clipping is enabled by default.

far distance. The distance from the far clipping plane to the origin of the VRC coordinate system.

fill area. An output primitive used to draw polygonal areas. The graPHIGS API supports the more general polygon primitive.

fill area with data. An output primitive consisting of a single contour, similar to the polygon primitive in the graPHIGS API element may include other information such as colors or normals that may be used to light and shade the primitive.

front face. Any portion of an area defining primitive whose geometric normal when transformed to NPC has a non-negative Z component. Front facing portions of primitives can be made invisible.

fixed size font. A text font in which every character has the same character body boundaries.

font designer. The person who designs text fonts for a graPHIGS API implementation. The font designer is responsible for choosing the font's aspects, including the justification points, using aesthetic criteria.

font directory. A resource of a graPHIGS API nucleus used to store font definitions supplied by a graPHIGS API shell.

frame buffer. An array of memory defining the color or intensity of each pixel on a workstation's display surface. Also see Indexed Frame Buffer and Component Frame Buffer.

frame buffer write protect mask. Consists of a set of bits defining which frame buffer bit planes are write protected.

G

generalized drawing primitive (GDP). A primitive structure element which is not supported on all workstation platforms.

generalized structure element (GSE). A structure element (typically a primitive attribute) which is not supported on all workstation platforms.

geometric normal. A vector that is perpendicular to the plane containing a primitive, facet, or sub-primitive. The geometric normal may be used in lighting calculations, and to determine whether primitives or portions of primitives are back facing.

geometric text. An output primitive for text display for which all text attributes are fully transformable. This is used for drawing text where the application requires explicit control over all aspects of the text's presentation.

graphics data. The collection of data required to produce a picture.

graPHIGS API description table. The data structure which gives the static (fixed) workstation-independent parameters and defaults of the graPHIGS API system.

graPHIGS APIstate list. The data structure which gives the dynamic (variable) workstation-independent control parameters of the graPHIGS API system.

global modeling transformation. During structure traversal, the composite modeling transformation of the parent structure, composed with the local modeling transformation to form the current composite modeling transformation.

H

hatch. One possible representation of the interior of a POLYGON primitive. The interior of the polygon is filled with a pattern of parallel and/or crossing hatch lines, selected from the workstation's hatch table.

hatch table. The workstation-dependent collection of possible hatches.

Hidden Line and Hidden Surface Removal (HLHSR). The process of displaying only those primitives which are not obscured by other primitives in the user's field of view. Also see Z-Buffer.

HLHSR. See Hidden Line and Hidden Surface Removal and Z-Buffer.

hierarchical. A relationship between structures resulting from the ability of structures to invoke other structures.

highlighting. An attribute indicating whether subsequent output primitives encountered during structure traversal are to be distinguished in some workstation-dependent manner. The current color used to highlight primitives is indicated by the highlight color index.

highlighting filter. A collection of highlighting classes (the Inclusion Highlighting Filter and the Exclusion Highlighting Filter) used to identify primitives which are eligible or ineligible for highlighting.

hollow. One possible representation of the interior of a POLYGON primitive. A boundary line only is displayed (the interior itself is empty).

HSV. The Hue, Saturation, Value color model.

I

intrinsic color. The color of an area defining primitive that is independent of light sources. This color is used as the diffuse color in reflectance calculations. In case lighting is disabled, the intrinsic color is used for rendering the primitive.

isoparametric curve. The parametric curve that is produced from a parametric surface by holding one of the independent variables constant.

image. A collection of image boards and possibly a color table that can be mapped to a view for display.

image board. A resource of a graPHIGS API nucleus which consists of an array of data used to store and display an image.

implementation-dependent. Some aspect of the system which cannot be completely specified, but will vary from implementation to implementation.

implicit regeneration. Regeneration of the picture on a display surface, which may occur when changes to the picture definition invalidate the displayed image. The application program did not explicitly request the regeneration.

inclusion highlighting set. Those highlighting classes whose primitives are declared eligible for highlighting.

inclusion invisibility set. Those invisibility classes whose primitives are declared eligible for invisibility.

inclusion pick set. Those pick classes whose primitives are declared eligible for picking.

indexed frame buffer. A type of frame buffer which contains one display color table index for each pixel on the display surface. Also see Frame Buffer and Component Frame Buffer.

individual attribute selection. Selection of one particular attribute value by a structure element.

infinite light. A type of light source that enters into the lighting calculation dependent on the orientation of surface being illuminated, but independent of the relative position of the surface being illuminated. This is used to simulate light sources relatively far from the surface being shaded. Same as directional light source.

inherit. The automatic passing of attribute selections from a parent structure to a child structure. The child structure need not specify all attributes explicitly, since the inherited values act as local defaults.

input class. A set of input devices that are logically equivalent with respect to their function. In graPHIGS API, the input classes are: Locator, Stroke, Valuator, Choice, Pick, and String.

input data record. A data structure containing detailed information about the measure and echo type of a logical input device. Provided by the application program when the device is initialized.

input device. A hardware capability used for input. A physical input device is either part of a display device or a separate part of the workstation.

input mode. One of the three possible means of requesting and obtaining data from a logical input device: REQUEST, SAMPLE, or EVENT.

input priority. One characteristic of a view; it helps determine the particular view whose inverse transformation is applied to locator input.

instancing. A method of defining an object to appear once in a data base and replicating it (without copying) multiple times with what may be different positions, sizes, orientations, and other attributes as inherited by the instance. (See also hierarchical and inheritance.)

interactive device. Any graphics device that supports both graphics output and input.

inquiry. The communication of data contained in a state list or description table to the application program. Can also be applied to other data structures such as structures (also called ″query″).

invisibility filter. A collection of invisibility classes (the Inclusion Invisibility Filter and the Exclusion Invisibility Filter) used to identify primitives which are eligible or ineligible for invisibility.

isotropic mapping. A transformation which preserves aspect ratio.

K

knot vector. A non-decreasing sequence of real numbers that is part of the definition of non-uniform B-splines. This vector consists of values for the independent variable of the parametric primitive, and is used in computing the B-spline basis polynomials.

L

label. A structure element consisting of an identifier which can be used to facilitate structure editing.

language binding. The binding of a functional specification to the syntax of a particular programming language.

lighting. The effect of light sources upon an area defining primitive.

lighting equation. A general formula through which the effect of lights illuminating area defining primitives is modelled.

light source. An entry in a workstation light source table. Four types of light sources are defined: ambient, infinite, positional and spot. All light sources have a color. Some light source types have a position, direction, concentration exponent, spread angle, and/or attenuation coefficients.

light source direction. A unit vector that defines the orientation of directional and spot light sources.

light source state. An attribute that selects which light sources in a workstation light source table are ON.

lighting. The effect of light sources upon an area defining primitive.

linetype. An attribute which indicates the style of the image of certain visible lines (e.g., solid, dashed). Values of this attribute apply only to primitives which create visible line segments (for example, POLYLINE). The default line type is SOLID.

linewidth scale factor. An attribute which indicates the relative width of the image of a visible line. The linewidth scale factor is applied to a workstation-dependent nominal value. Values of this attribute apply only to primitives which create visible line segments (for example, POLYLINE).

local modeling transformation. During structure traversal, the modeling transformation of the current structure. Composed with the global modeling transformation, possibly inherited from the parent structure, to form the current composite modeling transformation. Initialized to the identity transformation upon entering a structure.

locator. An input class providing world coordinate positioning information.

locator device. (1) A logical input device that provides a position in World Coordinates (WC); a *locator* logical input device can be implemented as a physical input device such as a mouse, a tablet, or a puck. (2) See also choice device, pick device, valuator device.

logical input device. An abstraction of one or more physical input devices which delivers logical input values to the application program. Logical input devices in graPHIGS API can be of class locator, stroke, valuator, choice, pick, or string.

logical input value. A value delivered by a logical input device.

M

marker. A glyph with a specified appearance which is used to identify a particular location. Markers are a means by which two-dimensional and three-dimensional points manifest themselves on the output display surface. Marker size is not subject to transformations, but position is.

marker size scale factor. An attribute controlling the size of a marker. The size is specified as a multiple of the nominal marker size, a workstation-dependent constant.

marker type. An attribute which defines the glyph used to denote the position of a visible POLYMARKER primitive.

measure. A value (associated with a logical input device) which is determined by one or more physical input devices and a mapping from the values delivered by the physical devices. The logical input value delivered by a logical input device is the current value of the measure.

measure process. A process in existence whenever a logical input device has been enabled for interaction. The current state of a measure process is the measure.

modal. A type of change mechanism which extends its effect until further notice, that is, until another change specification supersedes it. Attribute settings and control functions behave modally in graPHIGS API.

modal attribute settings. The specification of attributes during structure traversal in graPHIGS API using a current value which applies to subsequent output primitives until changed.

model. Refers to a computer's internal representation of an object in the form of an application program interleaved with graphics data. Your application uses the graphics data to build models that represents objects in the real world.

modeling coordinates. A local coordinate system in which graphical objects are defined by the application program using the graPHIGS API output primitives.

modeling coordinate system. A high-level system for defining (constructing) and manipulating objects. A modeling system describes objects to the graPHIGS API using modeling coordinates. Modeling coordinate objects are mapped into world coordinates using the modeling transformation.

modeling transformation. A transformation applied to all primitives (which are defined in modeling coordinate space), to position them in world coordinate space.

N

near clipping plane. A plane parallel to the view plane which is specified by a distance in viewing coordinate space from the origin along the N axis. When near plane clipping is enabled, portions of objects in front of the near clipping plane are discarded. Clipping against the near clipping plane is controlled separately from window clipping.

near distance. The distance from the near clipping plane to the origin of VRC.

near plane clipping. Near plane clipping is clipping against that part of the object which lies within and on the view volume, behind and on the near clipping plane. Near plane clipping is enabled by default. Near plane clipping is independent of far plane clipping.

new frame action. See Regeneration.

NIL. A value returned for some parameters of type ″enumeration″ (E) to indicate non-existence, for example, Inquire Open Structure returns NIL when no structure is open.

nominal value. The default, workstation-dependent value for an attribute such as linewidth, to which other possible values on that workstation are relative.

non-rigid transformation. Any transformation that doesn't preserve angles, and distorts the shape of objects, for example; perspective, shearing, and non-uniform scaling.

Non-Uniform Rational B-Splines (NURBS). A mathematically flexible means of defining parametric curves and surfaces.

normal. A vector that is perpendicular to a surface at a specific location. The normal is used to specify the orientation of a surface for use in the light calculation. Using the graPHIGS API, an application can approximate a surface by polygon with data primitives. In this case, the application could supply surface normals at each vertex to obtain a better visual representation. When this occurs, the normals are not necessarily perpendicular to the plane of the polygon.

nucleus. The part of the graPHIGS API which manages workstation, structure store, image board, and font directory resources. The nucleus is responsible for processing requests from attached graPHIGS API shells.

NURBS. See *Non-Uniform Rational B-Spline*.

O

object. Refers to both things and abstractions in the real world such as tangible substances, and intangible ideas and processes.

object-centered viewing. A viewing situation in which the object is considered fixed and the most likely dynamic behavior is for the viewer to move around the object. See *viewer-centered viewing*.

oblique view. A parallel projection whose projectors are not perpendicular to the view plane.

operator. User of an (interactive) application program which is implemented using the graPHIGS API The operator interacts with the application program through workstation input and output capabilities.

origin. A reference point whose coordinates are all zero.

output primitive. A basic graphical element (e.g., a line or a text string) used to construct an object. The values of attributes determine certain aspects of the appearance of an output primitive.

P

panning. The process of translating all elements of a picture to give the appearance of sideways movement to a point or object of interest. See *zooming*.

parallel projection. The image of a three-dimensional object in which lines that are parallel in the object appear parallel in the image without regard to relative distance or depth.

parent structure. A structure which invokes another structure, the child structure.

pattern. One possible representation of the interior of a POLYGON primitive. The interior of the polygon is filled with a two-dimensional array of color indexes, selected from the workstation's pattern table.

pattern table. The workstation-dependent table of possible pattern values.

perspective projection. The image of a three-dimensional object in which parallel lines appear to converge in the image as a function of relative distance or depth of the object from the position of the viewer.

PHIGS. The Programmer's Hierarchical Interactive Graphics System, a standard (ISO and ANSI).

physical input device. The actual hardware input device used to generate input to an application program. Physical input devices supply input data to logical input devices to generate the actual input to an application program. Also see logical input device.

pick. An input class providing identifying information (e.g., structure identifier, pick identifier, etc.) for a selected output primitive.

pick aperture. An implementation-dependent and workstation-dependent area for testing objects during a pick input operation. To be picked, at least some portion of the object must be within the pick aperture. The coordinate system used to define the pick aperture is implementation-dependent and workstation-dependent.

pick device. (1) A logical input device to select a graphical display element or segment; a *pick* logical input device can be implemented as a physical input device such as a screen cursor controlled by a mouse, a stylus, or a puck. (2) See also choice device, locator device, valuator device.

pick filter. A collection of pick classes (the Inclusion Pick Filter and the Exclusion Pick Filter) associated with a pick interaction, used to identify primitives which are eligible or ineligible for picking.

pick identifier. A ″name″ returned by the pick logical input device.

pick path. The traversal path to a picked primitive from its root structure, represented as a series of structure elements, pick ids, and structure ids.

pick path depth. The number of levels of the pick path returned on a pick, as specified when the pick device is initialized.

picture. The collection of information that is visible at any moment on a display surface.

pipeline. See transformation pipeline.

pixel. The smallest element of a raster display surface that can be independently assigned a color or intensity.

polygon. An output primitive consisting of a collection of polygonal areas which are considered as one area. Attributes are provided in the graPHIGS API to control the appearance of both the interior and/or the edges of the polygon.

polygonal area. A closed planar figure defined by the vertices of its linear boundary (or edge). The boundary may cross itself. Polygonal areas are defined using the POLYGON primitive.

polygon edge. The boundary of a polygon. Contained in the POLYGON primitive.

polyline. An output primitive consisting of a set of connected lines.

polyline set with data. An output primitive consisting of an unconnected set of polylines. The corresponding structure element may include other information such as colors that may be used to shade the primitive.

polymarker. An output primitive consisting of a set of markers.

positional Light. A type of light source that enters into the lighting calculation dependent on the orientation and relative position of the surface being illuminated.

priority. The priority used in displaying primitives, structures, and views.

primary trigger. The input trigger which terminates the measure process and causes the input to be returned to the application.

primitive. See Output Primitive.

projector. Lines which pass through each point of an object and intersect the view plane.

projection. A transformation function used to map three-dimensional coordinates to a constrained three-dimensional coordinate space. In particular, the intersections of all of the projectors with the view plane is called the projection of that object on the view plane. See also parallel projection and perspective projection.

projection plane. See View Plane.

projection reference point (PRP). A VRC point which determines the direction of all projectors in a parallel projection (i.e., all of the projectors are parallel to the vector defined by the projection reference point and the center of the window) or from which all projectors in a perspective projection emanate.

projection type. The type of projection to be used in view mapping. i.e., parallel projection or perspective projection.

prompt. Output to the operator indicating that a specific logical input device is available. Used to request input.

proportionally sized font. A text font in which each character's body boundaries vary as a function of that character's visible representation.

prune. To replace or bypass traversal of a portion of a structure network when a particular extent is completely clipped by either the modelling clipping limits, or the workstation transformation.

R

raster cell. See cell, pixel.

raster graphics. Computer graphics in which a picture is composed of an array of pixels arranged in rows and columns.

rational parametric polynomial primitives. Curves and surfaces that are defined via a vector polynomial that maps from parameter space to 4D homogeneous coordinates, and by a projection to 3D modelling coordinates.

realized value. An actual value used by a workstation depending on its capabilities. See SET Value.

reference. See Structure Reference.

reflectance equation. The equation that models the light reflected by an illuminated area defining primitive.

reflectance normal. A vector used in the reflectance calculation, typically derived from vertex normals or from the geometry of the primitive.

regeneration. The redrawing of the display. May be explicit or implicit. On some devices, it may also be automatic.

rendering. The process of generating the image of a geometric primitive based on the primitive's definition and attributes.

rendering color table. The workstation color table used to convert color index values into color vectors to be processed by the rendering pipeline. A rendering color table is sometimes referred to as an input color table because the values taken from the color table are used as input to the rendering pipeline. Also see display color table.

rendering pipeline. The sequential set of processes performed on a primitive to generate the primitive's image.

request input. One of three input modes. In request mode, the application program requests graphical input and then waits for an input value to be returned.

resolution. The number of visible distinguishable units in the device coordinate space. Contrast with addressable point.

RGB. The Red, Green, Blue color model.

right-handed system. A three-dimensional coordinate system in which, if the positive X-axis points up, then the positive Z-axis points toward the viewer. In mathematical terms, the cross product of the X unit vector with the Y unit vector produces the Z unit vector (Z = X x Y).

rigid transformations. Any transformation that preserves angles, so as to preserve the constant shape of objects and, more importantly, preserves normals; for example rotations, translations, and uniform scales.

root structure. A structure which has been associated to a workstation and view or both.

rotation. Turning an object about an arbitrary axis in a two-dimensional or three-dimensional coordinate space.

S

sample input. One of three input modes. In sample mode, the application program retrieves the measure of a logical input device without waiting for any external operator action.

scaling. Enlarging or reducing the size of an object by multiplying the coordinates of its output primitives by a constant value.

secondary trigger. An input trigger which performs some intermediate function in the accumulation of input during the measure process. For example, this may involve adding, deleting, or modifying the current data in the input device buffer.

set value. A value as provided by the application program to a workstation. The workstation may not be able to render the SET value exactly.

shading. The interpolation portion of the rendering pipeline. Shading is applicable to area defining primitives.

shell. The part of the graPHIGS API which interfaces with an application program. The shell performs parameter checking, error handling, event handling, and the building of data buffers containing nucleus requests.

silhouette edge. A curve on a surface that exhibits a special relation to an observer of the surface. A point on the surface belongs to the silhouette edge if the surface normal at that point is perpendicular to the vector emanating from the point to the eye point.

solid. One possible representation of the interior or a polygon primitive. The interior of the image is filled with the solid color specified by the interior color index attribute.

specular color. The color of specular highlights of an area.

specular exponent. A number representing the shininess of a surface. The higher the specular exponent, the shinier the surface. Negative exponents are not permitted.

specular reflection. An approximation of the light reflected from a surface which causes highlights on shiny objects. The intensity of specular reflections, unlike diffuse reflections, is highly dependent on the viewing angle of the observer.

specular reflection coefficient. The fraction of light from non-ambient sources specularly reflecting from an area.

spot light. A type of light source like a positional light, but that also restricts the zone of illumination to a semi-infinite cone and/or concentrates the brightness of the light along a ray emanating from the light source position.

spread angle. An angle that determines the cone of influence of a spot light source. Only points inside this cone are affected by the corresponding light source.

state list. A data structure containing current values of system variables. Application programs can obtain state list contents through inquiry. See graPHIGS API State List, Workstation State List, Structure State List, Error State List, Utility Function State List, Traversal State List.

string. An input class which returns a character string.

string device. A logical input device of input class string.

stroke. An input class which returns a sequence of positions in world coordinates.

stroke device. A logical input device of input class stroke.

structure. A grouping of elements which provides the necessary information to describe all or part of an object, excluding workstation state and control information.

structure element. A unit of data which comprises a structure. Structure elements include output primitives, attribute selections, labels, application data, pick set specifications, transformation selections, and structure invocations.

structure identifier. The identifier by which a structure can be referenced. An integer in the graPHIGS API.

structure invocation. The process of executing another structure.

structure network. A hierarchical family of related structures.

structure reference. A reference to a structure; created by executing a structure or by posting it as a root.

structure state list. A data structure containing information for all structures defined during the execution of a graPHIGS API program.

structure store. A resource of a graPHIGS API nucleus used to store structures.

structure traversal. The processing of structures to produce an image.

structure query. Communication of information in the current structure element to the application program, at the request of the application program.

sub-primitive. An implementation dependent fragment that may be generated by the tessellation of a curve or surface primitive that is rendered in much the same way a polyline or polygon is rendered.

surface properties. An attribute that defines the set of reflectance coefficients and other aspects which provide information about an area defining primitive to the lighting and shading stage of the rendering pipeline. Surface properties include ambient reflection coefficient, diffuse reflection coefficient, specular exponent and transparency coefficient.

T

tessellation. The process of subdividing a curve or surface primitive into lines or polygons respectively.

text. An output primitive consisting of a character string.

text alignment. An attribute of text which specifies the mode of justification. This attribute has two components, one for horizontal justification (LEFT, LEFT BODY, CENTER, RIGHT BODY, RIGHT) and one for vertical justification (TOP, CAP, HALF, BASE, BOTTOM).

text font. A workstation-dependent attribute of text which indicates certain aspects of a visible character (such as style, typeface, boldface, italic, polygonal vs. outline, etc.). Text font in graPHIGS API is an integer, selecting a font encompassing these aspects, as defined by the font designer. Text font is one of the components of text bundle entry.

text path. An attribute of text which indicates the direction, relative to the character up vector, in which the text is drawn within the text plane.

text plane. The plane on which text appears.

text position. A point defined within the text primitive which (together with the text reference points) defines the text plane. The text position also determines (together with text alignment), the location of the text string.

text precision. A workstation-dependent attribute of text which indicates the accuracy of interpretation of positioning and clipping the text primitive.

text reference points. Two points in the text function which (together with the text position) determine the text plane.

transformation. A mathematical operation which converts the coordinates of one representational system into another; for example, modeling coordinates to world coordinates.

transformation pipeline. The series of mathematical operations which act on output primitives and geometric attributes to convert them from modeling coordinates to device coordinates.

translation. The application of a displacement to the position of an object without rotation or scaling.

transparency. An attribute applied to surface primitives which permits primitives behind the surface to be visible through the surface.

transparency coefficient. The fraction of light transmitted through an area.

traversal. See structure traversal.

traversal state list. A data structure containing information for all structures traversed during the execution of a graPHIGS API program.

trimmed surface. A primitive that is defined by a parametric surface and a set of contours in parameter space. The contours specify the portions of parameter space over which the surface is rendered.

trimming curve. A parametric curve defined in the parameter space of the surface to which it applies. Trim curves are combined to form contours which limit the parameter range over which the corresponding surface is rendered.

trigger. A physical input device or set of devices which an operator can use to indicate completion of a measure process.

trigger process. A process which notifies a recipient logical input device when a trigger is fired. The trigger process only exists when a device is in EVENT mode, or is in REQUEST mode with a request pending.

U

utility function. One of a group of functions provided to create a graPHIGS API matrix from the parameters which are natural to the application program. Functions are provided to assist with the creation of both modeling and viewing matrices. Viewing parameters are used which support a commonly used viewing approach. Application programs may choose to use another approach to viewing, or compose the matrices directly for other reasons.

utility function state list. The data structure which contains the parameters used by the utility functions which support the creation of viewing matrices.

UVN Coordinate System. See Viewing Coordinates.

V

valuator device. (1) A logical input device that provides a scalar value; a *valuator* logical input device is typically implemented as a physical input device such as a dial. The 5080/6090 dials unit with eight dials is an example; each of the eight dials is a valuator input mechanism. (2) See also choice device, locator device, pick device.

vector. A straight line segment which has both magnitude and direction. Vectors are used in the graPHIGS API to specify certain parameters. In some cases, only the direction of the vector (not its magnitude) is actually used by the graphics system (e.g., character up direction).

vertex color. A Color associated with each vertex of some primitives such as polyline set with data, fill area with data, fill area set with data, polyhedron with data, or triangle strip with data. The color is contained within the primitive definition and may be used as input to the rendering pipeline.

vertex normal. A vector supplied with an area defining primitive and associated with a vertex of such a primitive. Vertex normals are typically used for determining reflectance normals. For example, an application can approximate a surface using polygon 3 with data primitives, and supply vertex normals to obtain a better visual representation. In this case, the normals are not necessarily perpendicular to the plane of the polygon, but might be normal to the surface being approximated.

view. The description of a viewing operation which produces a displayable picture from world coordinates. Each workstation has a table of views which is accessed in a workstation-independent manner.

view distance. See view plane distance.

viewer-centered viewing. A viewing situation in which the viewer's position is considered fixed and the most likely dynamic behavior is for the viewer to ″look around″ at the world.

viewing coordinates. A three-dimensional coordinate system used for defining windows and projection parameters. It is defined by its view orientation matrix. The axes of the viewing coordinate system are labeled U, V, N.

viewing parameters. The parameters which may be used to define the viewing coordinate system and other aspects of a view.

viewing pipeline. See transformation pipeline.

viewing transformation. A normalization operation which maps positions in world coordinates to positions in viewing coordinates. In addition, a viewing operation specifies the portion of world coordinate space that is to be displayed.

view mapping. The transformation from viewing coordinates to logical device coordinates. This transformation includes a projection mapping and optional clipping. Also called the window or viewport transformation.

view matrix. A matrix used to specify the viewing transformation. The view matrix may be composed using utility functions.

view plane. A two-dimensional plane onto which three-dimensional objects are projected.

view plane distance. The distance of the view plane from the origin of VRC. Used by the viewing utilities.

view plane normal. A vector normal to the view plane used to orient the plane. Used by the viewing utilities.

viewport. An application program specified part of normalized projection coordinates. The contents of the view volume are mapped to the viewport.

view reference point. A convenient world coordinate point on or near the object being viewed. All of the viewing parameters are specified relative to the view reference point. The view reference point becomes the origin of the viewing coordinate system. Used by the viewing utilities.

view surface. See display surface.

view table. The workstation-dependent table of views found on each workstation.

view up vector. A vector in world coordinates relative to the view reference point which, if it were within the window, would appear upright on the display surface. The view up vector is projected onto the view plane, in the direction of the view plane normal; this projection (which is the U axis in the viewing coordinate system) orients the world coordinate ″up″ direction for the view. The default view up vector is (0,1,0) which causes the Y axis to be ″up″ Note that this default will not work if the view plane normal is parallel to the Y axis. Used by the viewing utilities.

view volume. The view volume is either a (possibly truncated) pyramid for perspective projection or a volume with sides parallel to the direction of projection for parallel projection. The boundaries of the view volume are determined by the window defined in the view plane, the near and far clipping planes, and projections from the projection reference point through the window corners (for perspective projection) or projectors in the direction of projection through the window corners (for parallel projection). If the projection reference point is not on the view plane normal, a sheared projection will result. In strictly two-dimensional applications, the view volume reduces to an area on the view plane.

visibility. An attribute which indicates whether output primitives are actually visible on the display surface. In the graPHIGS API, visibility is controlled using class names and filters.

W

Ward. A section of a Double-Byte Character Set where the first byte of all the codes belonging to it are the same. A Ward has a unique number which represents the first byte of the code belonging to that Ward.

window. A rectangle on the view plane which defines the XY limits of the view volume.

window clipping. Clipping against the sides of the view volume defined by the window.

window or viewport transformation. See view mapping.

wire frame. A three-dimensional object represented as a series of line segments outlining its surface.

workstation. An abstraction of physical graphics devices providing the logical interface through which the application program controls physical devices.

workstation attribute. Those attributes which have values that are specific to a workstation. Workstation attribute values are workstation-dependent.

workstation attribute tables. A conceptually upper boundless table of all workstation attribute table entries for a given table-driven workstation attribute, including, but not limited to, the bundled attributes. The attribute tables are referenced indirectly through an index, which may be either workstation-independent (e.g., Text Index) or workstation-dependent (e.g., Text Color Index). Not all workstation attributes are table-driven. The workstation tables are workstation-dependent since they exist for each workstation.

workstation attribute table entry. One of a set of specifications for a particular workstation attribute. For example, a color table entry is a workstation attribute table entry since it is a specification of one of a set of colors. Format is of no concern to the user.

workstation attribute table index. A reference to a workstation attribute table entry. The workstation attribute table index is a global-modal attribute. The index can be either workstation-independent (e.g., Text Index) or workstation-dependent (e.g., Text Color Index).

workstation attribute registers. Conceptual registers in which the current value of each workstation attribute is kept during structure traversal. These values determine the actual appearance of the image. See *conceptual registers.*

workstation category. Determines whether a particular workstation type can process graphics input only, graphics output only, or both output and input.

Workstation Configuration Variable (WCV). Items describing the capabilities of a workstation which may vary depending on its physical configuration are identified as WCV's.

workstation-dependent. Entities that can vary from workstation to workstation.

workstation description table. The data structure which gives static (fixed) information for a particular workstation type.

workstation identifier. Uniquely identifies a particular workstation.

workstation-independent. Entities which are the same across all workstations.

workstation state list. The data structure which gives dynamic (variable) information for a particular workstation.

workstation transformation. An isotropic transformation which maps the boundary and interior of a workstation window into the boundary and interior of a workstation viewport, performing translation and scaling. This transformation maps normalized projection coordinates (NPC) to device coordinates. The effect of preserving aspect ratio is that the interior of the workstation window might not map to the whole of the workstation viewport.

workstation type. A type or class of actual workstation, sharing common characteristics and a single workstation description table.

workstation viewport. A portion of display space currently selected for output of graphics.

workstation window. The area in NPC space which is represented on a display surface and is mapped to the workstation viewport.

World Coordinates (WC). Device-independent Cartesian coordinates used by the application program to organize modelled two-dimensional or three-dimensional objects for display. The effect of applying the composite modeling transformation to modeling coordinates is to produce world coordinates.

Z

Z-Buffer. A hardware feature that permits a workstation to perform Hidden Line and Hidden Surface Removal (HLHSR).

zooming. The process of scaling all elements of a picture to give the appearance of having moved towards or away from a point or object of interest.

Appendix D. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:
(c) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (c) Copyright IBM Corp. _enter the year or years_. All rights reserved.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
AIXwindows
GDDM
IBM
RS/6000

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be the trademarks or service marks of others.

Readers' Comments — We'd Like to Hear from You

The graPHIGS Programming Interface: Understanding Concepts

Publication No. SC33-8191-04

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name _____ Address _____

Company or Organization _____

Phone No. _____

Readers' Comments — We'd Like to Hear from You
SC33-8191-04

IBM

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Information Development
Department H6DS-905-6C006
11501 Burnet Road
Austin, TX 78758-3493

Fold and Tape | **Please do not staple** | Fold and Tape

SC33-8191-04

Cut or Fold
Along Line

Printed in U.S.A.

SC33-8191-04