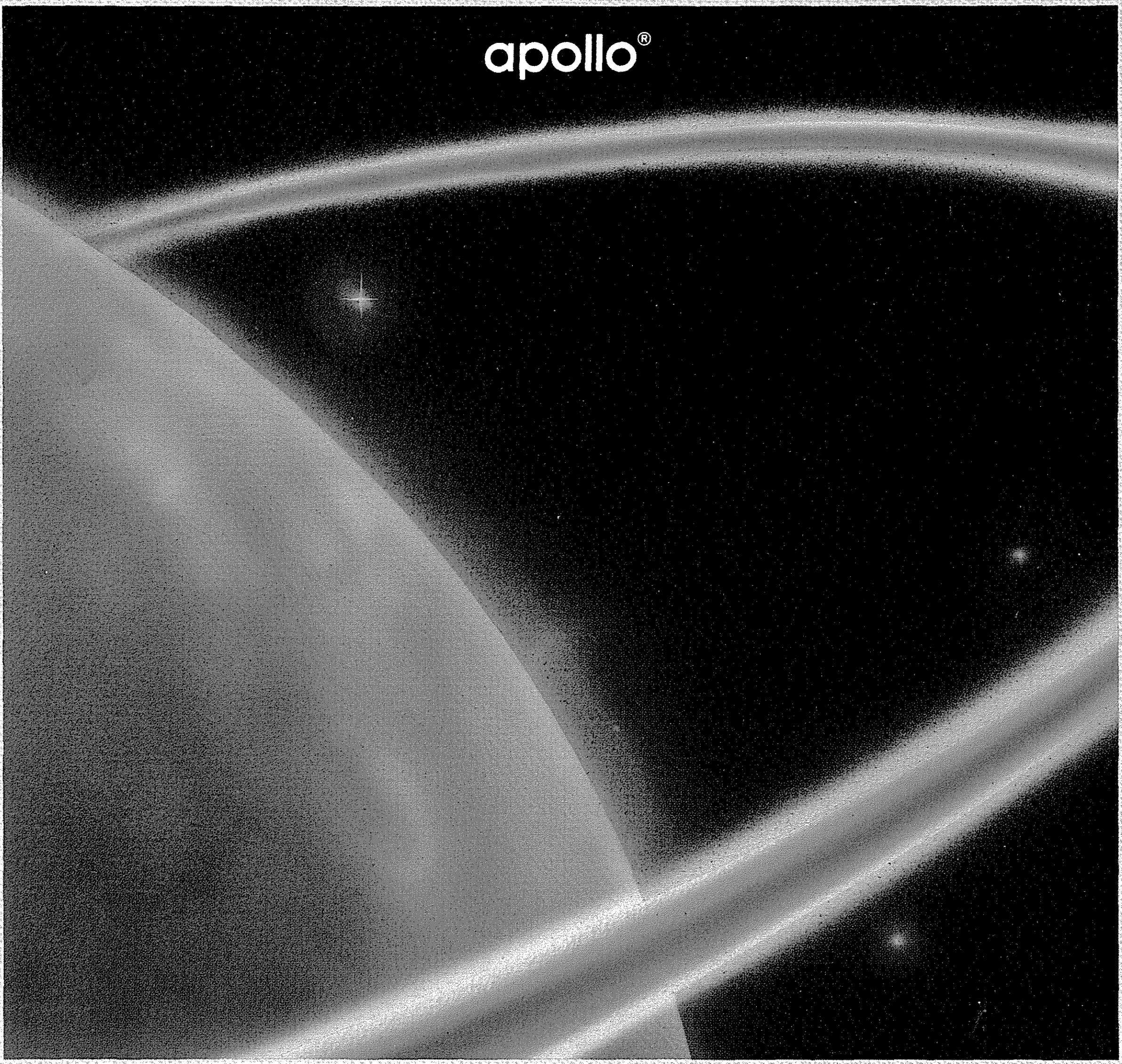
apollo®



D O M A I N

Order No. 005097
Revision 00
Software Release 9.0

Programming With DOMAIN 2D Graphics Metafile Resource

Programming with DOMAIN 2D Graphics Metafile Resource

Order No. 005097
Revision 00
Software Release 9.0

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: July, 1985

Preface

Programming With DOMAIN 2D Graphics Metafile Resource describes the DOMAIN® two-dimensional graphics metafile resource system (2D GMR)and explains how to use this system in developing graphics application programs. Detailed descriptions of the user-callable routines for the DOMAIN 2D Graphics Metafile Resource are included in the *DOMAIN System Call Reference*, Volume 1.

Audience

This manual is for programmers who use the DOMAIN 2D Graphics Metafile Resource to develop application programs. It is assumed that users of this manual have some knowledge of computer graphics and have experience in using the DOMAIN system.

Organization of this Manual

This manual contains fifteen chapters and six appendices. Cross-references in the manual indicate the chapter number and the section number. For example, a reference to Section 3.6 refers to Chapter 3 and Section 3.6 in the chapter.

Chapter 1	Presents an overview of the graphics metafile package and a comparison with other DOMAIN graphics packages.
Chapter 2	Describes the DOMAIN display, the 2D GMR™ bitmap, and the effect of initialization mode on the display of graphic images. Coordinates systems are defined.
Chapter 3	Presents the structure of 2D GMR application programs, including controlling files and segments and instancing segments. The chapter concludes with a basic sample program.
Chapter 4	Describes the draw and fill primitives and explains how to insert them in segment. The procedure for displaying all or part of a file or segment is described. Instancing and transformation routines are presented with a program to illustrate their use.
Chapter 5	Describes the use of individual attribute commands, explains how to use attributes in relation to instancing, and provides a program to illustrate these functions.
Chapter 6	Explains how to insert text and text attributes into a segment. Font families and their use are described, along with techniques for creating stroke fonts. Programs illustrate the use of routines.
Chapter 7	Describes the use of the primary segment in displaying a metafile. The chapter also includes discussion of segment characteristics and coordiantes data types.
Chapter 8	Describes the display environment and coordinate systems used with the graphics metafile package. Viewing routines are presented with a sample program to illustrate their use.

Chapter 9	Provides an overview of functions used for interactive editing. The routine for changing the editing mode is described and illustrated with a sample program.
Chapter 10	Describes the following interactive functions: replacing commands, establishing a refresh state, picking commands and segments, using input operations and event reporting, controlling the cursor, and reading a metafile. Program examples illustrate many of these functions.
Chapter 11	Explains how to extend the 2D GMR to include GPR™ routines and user-defined primitives. This extension to the 2D GMR package is illustrated with a sample program.
Chapter 12	Describes the routines and the external file format used in generating hard-copy output of graphics data.
Chapter 13	Describes the use of attribute classes and blocks, and explains how to tie attribute blocks to attribute classes for the entire display and for individual viewports. Programming examples illustrate the use of attribute routines.
Chapter 14	Describes advanced display techniques including using color and changing the viewport border and background. Programming examples illustrate these techniques.
Chapter 15	Describes the use of tags. The chapter also presents techniques for optimizing performance in using 2D GMR. Relationships of the DOMAIN graphics packages are discussed.
Appendix A	Presents a glossary of graphics terms in relation to the 2D GMR package.
Appendix B	Illustrates the 880 and low-profile keyboards and keyboard charts.
Appendix C	Presents an example of a program that prints out the entire contents of a metafile in readable form.
Appendix D	Contains a Pascal program with attributes and instancing.
Appendix E	Contains the programming examples presented in the manual translated into C.
Appendix F	Contains the programming examples presented in the manual translated into FORTRAN.

Additional Reading

For information about using DOMAIN Graphics Primitives, see *Programming With DOMAIN Graphics Primitives* (order no. 005808) and the *DOMAIN System Call Reference*, Volume 1 (order no. 007196) and Volume 2 (order no. (007194).

For information on DOMAIN Core Graphics, see *Programming with DOMAIN Core Graphics* (order no. 001955).

For information about using the DOMAIN system, see the *DOMAIN System Command Reference* (order no. 002547). For information about the software components of the operating system and user-callable system routines, see the *DOMAIN System User's Guide* (order no. 005488). For language-specific information, see the *DOMAIN FORTRAN User's Guide* (order no. 000530), *DOMAIN C Language Reference* (order no. 002093), and *DOMAIN Pascal Language Reference* (order no. 000792). For information about the high-level language debugger, see the *DOMAIN Language Level Debugger Reference* (order no. 001525).

On-Line Sample Programs

The programs from this manual are stored on-line, along with sample programs from other DOMAIN manuals. We include sample programs in Pascal, C, and FORTRAN. All programs in each language have been stored in master files (to conserve disk space). There is a master file for each language.

To access any of the on-line sample programs, you must create one or more of the following links:

```
For Pascal examples:  $ CRL ~COM/GETPAS /DOMAIN_EXAMPLES/PASCAL_EXAMPLES/GETPAS

For C examples:       $ CRL ~COM/GETCC  /DOMAIN_EXAMPLES/CC_EXAMPLES/GETCC

For FORTRAN examples: $ CRL ~COM/GETFTN /DOMAIN_EXAMPLES/FTN_EXAMPLES/GETFTN
```

To extract a sample program from these master files, all you have to do is execute one of the following programs:

```
To get a Pascal program:     $ GETPAS

To get a C program:          $ GETCC

To get a FORTRAN program:   $ GET FTN
```

These programs will prompt you for the name of the sample program and the pathname of the file to copy it to. Here is a demonstration:

```
   $ GETPAS
   Enter the name of the program you want to retrieve -- STAR_MOVE
   What file would you like to store the program in? -- STAR_MOVE.PAS

   Done.
   $
```

You can also enter the information on the command line in the following format:

   $ GETPAS <name_of_program_to_retrieve> *name_of_output_file*

For example, here is an alternate version of our earlier demonstration:

```
   $ GETPAS STAR_MOVE STAR_MOVE.PAS
```

GETPAS, GETCC, and GETFTN will warn you if you try to write over an existing file.

For a complete list of on-line DOMAIN programs in a particular language, enter one of the following commands:

```
$ GETPAS HELP
$ GETCC HELP
$ GETFTN HELP
```

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

UPPERCASE	Uppercase words or characters in formats and command descriptions represent commands or keywords that you must use literally.
lowercase	Lowercase words or characters in formats and command descriptions represent values that you must supply.
[ ]	Square brackets enclose optional items in formats and command descriptions. In sample Pascal statements, square brackets assume their Pascal meanings.
{ }	Braces enclose a list from which you must choose an item in formats and command descriptions. In sample Pascal statements, braces assume their Pascal meanings.
CTRL/Z	The notation CTRL/ followed by the name of a key indicates a control character sequence. You should hold down the <CTRL> key while typing the character.
< >	Angle brackets indicate a key to be pressed.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments. By using these formal channels you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference* manual. Refer to the CRUCR (Create User Change Request) Shell command description. You can view the same information on-line by typing:

```
$ HELP CRUCR <RETURN>
```

For your comments on documentation, a Reader's Response form is located at the back of this manual.

Contents

Illustrations

Tables

Chapter 1
Introduction

The DOMAIN 2D Graphics Metafile Resource package (hereafter referred to as 2D GMR) and this manual are intended for programmers who wish to develop graphics applications packages. The DOMAIN 2D GMR package provides a versatile, efficient tool for developing a graphics applications system that stores and displays picture data.

The purpose of this manual is to present concepts, procedures, and examples for users of the 2D GMR package. For a complete description of the user-callable routines of the 2D GMR package, see Volume 1 of the *DOMAIN System Call Reference*. The information in this manual is intended for programmers with some familiarity with computer-based graphics. The explanations and examples are provided for programmers with limited experience, as well as for those who have worked extensively with computer graphics.

1.1. What 2D GMR Provides the Application Developer

The 2D GMR package is a collection of routines that provide the ability to create, display, edit, and store device-independent files of picture data. The package provides routines for developing and storing picture data and displaying the graphic output of that data. The 2D GMR package provides you with the necessary support to build a graphics system "with a memory." The package integrates graphics output capabilities with file handling and editing capabilities.

The standard form of data storage in this package is a metafile. A metafile is a device-independent collection of picture data (vector graphics and text) that can be displayed. The metafiles you create are stored and available for you to redisplay, revise, and reuse. They are not static copies of display bitmaps; rather, metafiles contain lists of commands used to build a graphic image.

The Table 1-1 summarizes the capabilities of the 2D GMR package and explains what these enable you to do.

Table 1-1. Capabilities of the 2D GMR Package

2D GMR Capability:	Enables you to do this:
STORAGE	
Graphics system with a memory	Integrate graphic data, nongraphic data, display characteristics, editing, file storage, and hard-copy output.
Virtual storage of metafiles	Store files up to 256 megabytes.
MODELING/VIEWING	
Commands	Describe the least divisible element of a picture.
Segmentation	Group commands that make up separate items of a picture; name the items; reuse the items.
Nested segmentation	Have items include other items.
Instancing	Use a single sequence of commands multiple times with different transformations and attributes applied.
Scaling	Make the displayed picture larger or smaller.
Translation	Move the displayed picture.
Multiple viewports	Look at more than one part of the picture simultaneously; make changes and see the change in each view.
Flexibility in data types	Supply coordinate data as 16- or 32-bit integers or as single-precision real numbers.
A range of commands for drawing and filling	Draw lines and arcs; draw and fill circles and polygons
Attributes	Establish characteristics such as line style and background before and during display.
Blocks of attributes	Create a data structure that holds a collection of values that specify attributes.

Table 1-1. Capabilities of the 2D GMR Package (continued)

EDITING	
Editing commands within segments while viewing	Create interactive application easily; change picture details interactively
Identifying and picking segments and commands	Choose the focus of interactivity
INPUT/OUTPUT	
Accepting coordinate input from logical devices	Use input devices such as a mouse or puck with easy interface
Output to external file	Transfer data to hard-copy devices

Within a metafile, commands are grouped into segments. Each segment is a named entity consisting of a sequence of commands. A segmentcan be referred to from another segment, in a manner analogous to a subroutine call.

Individual commands within segments of the metafile describe the least divisible components of the picture. Commands are categorized as primitive commands, attribute commands, instance commands, and tag commands. These commands are defined as follows:

- *Primitive commands*: Describe the single least divisible, displayable components of a picture. These components are, for example, polylines (lists of linked line segments), rectangles, circles, and text.

- *Attribute commands*: Contain values that specify the manner in which components of the picture are to be drawn, for example, the line style or text size. Attribute values may be modified individually or in blocks.

- *Instance commands*: Cause references to be made to other segments. Instancing allows multiple uses of a single sequence of commands, with different transformations applied.

- *Tag commands*: Provide comments within the file that do not affect the picture.

Every command is part of some segment. There are no commands outside of all segments.

Applications programs call 2D GMR routines to edit and display files. These routines are categorized into modeling routines and viewing routines:

- *Modeling routines*: Control and edit metafiles.

- *Viewing routines*: Affect the form in which picture data within metafiles is displayed.

Please note our usage of the words "command" and "routine:"

- A command is a part of a metafile.
- A routine is a procedure or function which operates on metafiles.
- The 2D GMR package is a collection of routines that can edit commands within metafiles, or can affect how these commands are to be displayed. Each command is a single element of a picture as stored in the metafile.

The distinction between the 2D GMR package and the metafiles it creates is summarized in Table 1-2.

Table 1-2. 2D GMR Package and the Metafile

```
          DOMAIN 2D Graphics Metafile Resource

 A collection of routines that the programmer can call:

     Modeling Routines:   Control and edit Metafiles

      Viewing Routines:   Affect how the picture data
                          in the metafile is displayed
---------------------------------------------------------
          A graphics metafile contains segments.

          A segment contains commands.

          A command is a primitive command or

                       an attribute command or

                       an instance command or

                       a tag command.
```

You call modeling routines to affect the state of the metafile package, or to affect the contents of the files. Modeling routines create, open, and close files and segments, and insert, read, copy, and delete commands within segments. For each type of command that can occur in a metafile, a routine is provided to insert that command into the file; another routine is provided to read the parameters of that command from the file.

Using viewing routines, you may display the images produced by the data in a file. You can then edit the file and display the revised image. In all display modes except one (within-GPR), coordinates are device-independent. This independence allows convenient display of the output of the file (or regions of it) on the screen, or on another device such as a printer. The exception to device independence is the display mode called within-GPR.

Data from input devices, such as a touchpad or a mouse, may be processed and used to help build files.

The 2D GMR package does not operate directly on bitmaps (unlike other DOMAIN graphics packages). Instead, the 2D GMR routines modify either the contents of a metafile or the manner in which the metafile is displayed. The 2D GMR package then handles the conversion of the metafile to a bitmap for display or for hard-copy output (see Figure 1-1).

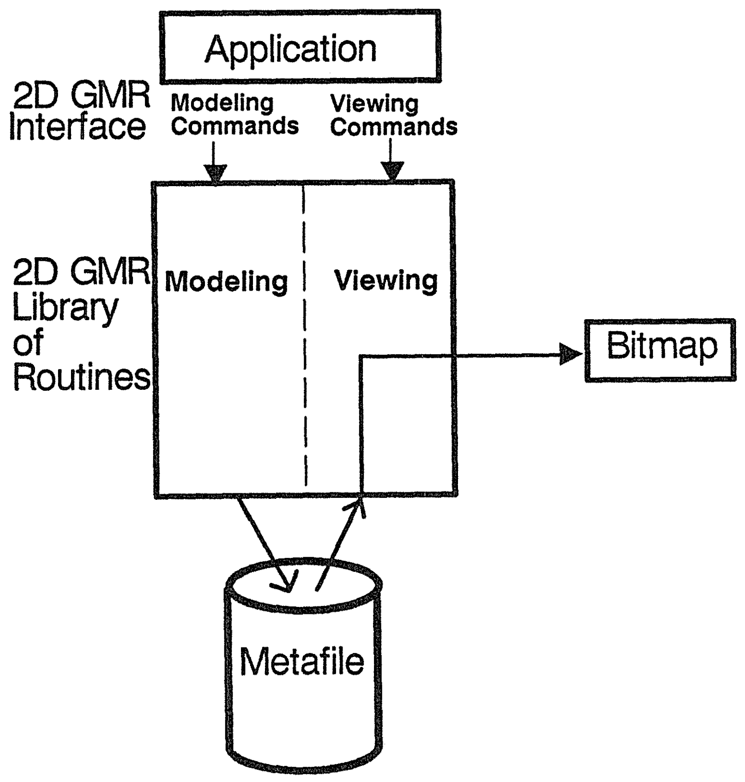


Figure 1-1. 2D GMR Library of Routines

The 2D GMR package is a library primarily containing two types of routines: modeling and viewing. These two types of routines operate on metafiles as follows:

Modeling routines modify the metafile. Viewing routines read the metafile and display it based on viewing parameters.

1.2. Data Types

For flexibility in modeling and for speed in storing data, you may supply coordinate data to the 2D GMR package in three formats:

- 16-bit integers
- 32-bit integers
- Single-precision real numbers

Different routines exist to accept data in these different forms, as described in Section 7.3. You may supply data in different formats to different commands within a segment.

1.3. Storage and Display

The 2D GMR package manages both storage and display of picture data. It differs from graphics packages that do not store an image for later display in the same way that a word processor differs from a typewriter.

The structure of the metafile package is analogous to the structure of a word processor or text editor, as outlined in Table 1-3. Modeling routines edit metafiles by inserting, changing, and inquiring about commands within the metafile. They let you build a file, as when you type characters into a text file that is open for editing.

Viewing routines control the form in which metafiles are displayed. These routines let you look at a file, but change only how it is displayed. This is similar to the commands <MOVE> and <GROW> used for a window. These routines do not change the contents of a file, but they change such characteristics of the displayed image as placement and size.

Table 1-3. Graphics Metafile Package and a Word Processor

2D GMR Package	Word Processor
The package manages files of picture data.	The processor manages files of text.
Files contain segments.	Files contain lines.
Segments contain commands.	Lines contain characters.
Some modeling routines add commands to a file.	Typing alphanumeric keys adds characters to a file.
Some modeling routines delete commands or copy segments.	Some control keys delete characters or copy lines.
Viewing routines change what part of the file is displayed.	Some control keys change what part of the file is displayed.

1.4. Extendable Package

The 2D GMR package is extendable. This means that you can use additional routines to mix 2D GMR routines calls and GPR routines. The 2D GMR package provides two options to allow you to expand it for your application.

- *Within-GPR Mode*: A subset of 2D GMR routines can operate in conjunction with GPR applications. This allows you to use 2D GMR files within a GPR-based application. Within-GPR mode can provide 2D GMR display advantages without rewriting existing GPR-based user interfaces. This mode also provides a migration path from GPR to 2D GMR (see Chapter 11).

- *User-Defined Commands*: Commands that you define within 2D GMR allow you to specify additional display routines for commands that you define.

1.5. Graphics Metafiles and Other DOMAIN Graphics Packages

The DOMAIN system also has two other graphics packages: DOMAIN Graphics Primitives (GPR) and DOMAIN Core Graphics. The graphics primitives library is built into your DOMAIN system. The routines (primitives) that make up the library let you manipulate the least divisible graphic elements to develop high-speed graphics operations. These elements include lines and polylines, text with various fonts, and pixel values. For a detailed description of graphics primitives, see the *Programming with DOMAIN Graphics Primitives* and the *DOMAIN System Call Reference*.

The DOMAIN system also has an optional Core graphics package. The Core graphics package provides a high-level graphics environment in which to build portable graphics application systems. For a detailed description of Core graphics, see the *Programming with DOMAIN Core Graphics*.

For a description of some of the most significant differences between 2D GMR and GPR, and betweeen 2D GMR and Core, see Chapter 15. The distinctive characteristics of the three systems are as follows:

- *Graphics Metafiles*: Commands are stored in device-independent files of picture data. The 2D GMR system lets you create, edit, display, and store picture data. Storage and rapid redisplay functions are combined into one package. This allows rapid interactive editing and redisplay. Coordinates are device-independent, providing flexibility in the development and use of application programs.

- *Graphics Primitives*: The function calls cause changes to be made to a bitmap. There is no memory of the calls performed except to the limited extent of being able to save a static image at any given time. Storing the bitmap in a file does not save the sequence of graphics commands that were used to create that bitmap. Therefore, redrawing usually requires that an application program itself keep track of and reexecute the calls. GPR display coordinates are device-dependent.

- *Core Graphics*: The functions in this package conform to an industry standard. The functions include modeling and viewing capabilities. The Core package stores segments only for redisplay during the same session; no permanent copy is created. These segments cannot contain instances of other segments. Coordinates are device-independent, providing flexibility in the development and use of application programs.

The 2D GMR package is distinct from the graphics primitives (GPR) package in this way: GPR operations are performed directly to the output device, while 2D GMR operations read, modify, or display a metafile (see Figures 1-2 and 1-3). The 2D GMR package initializes the GPR package for graphics display purposes; however, the use of 2D GMR and GPR commands within a single program is allowed in only one 2D GMR display mode. This mode is discussed in Section 11.1.2.

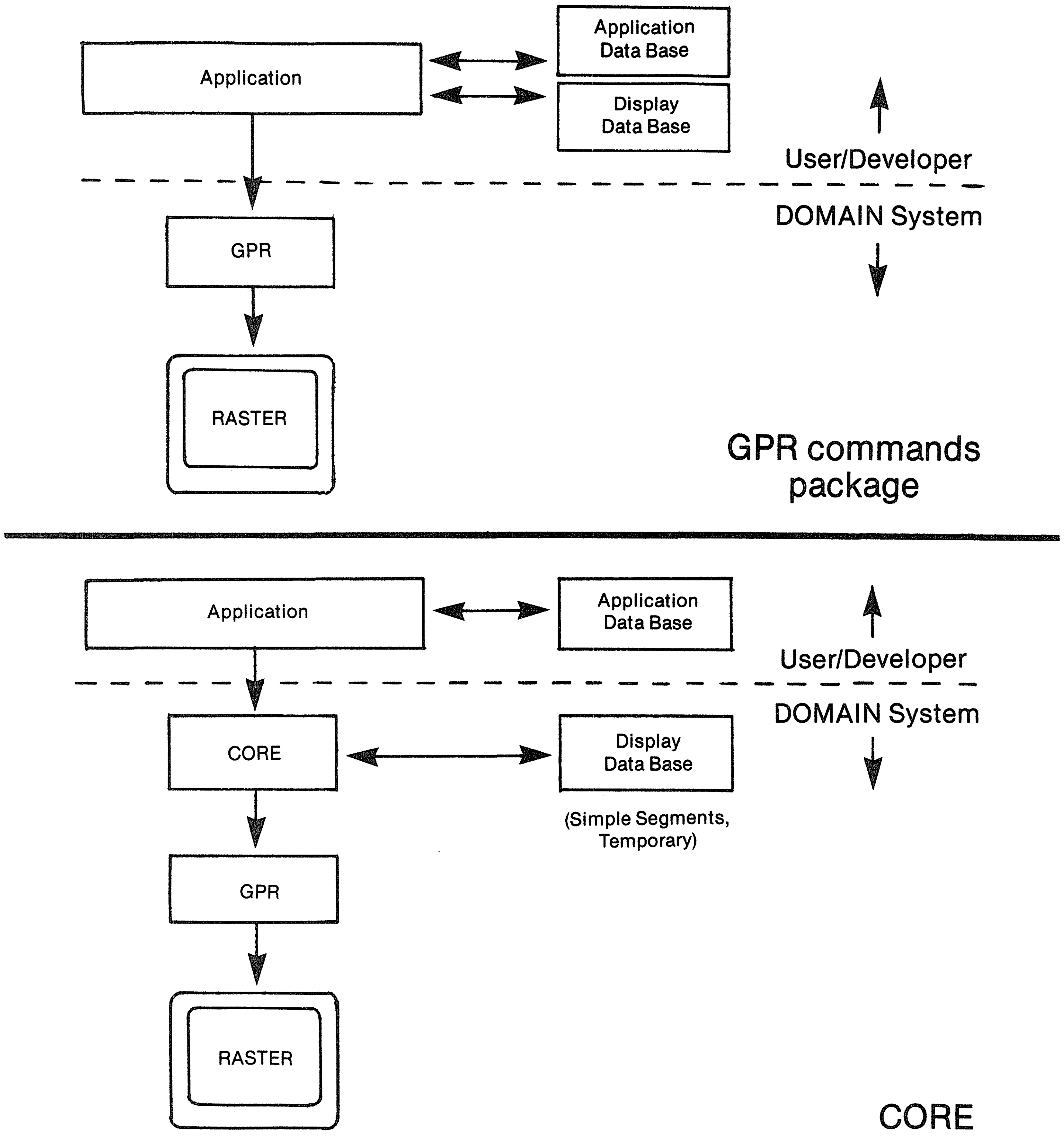


Figure 1-2. Relationship of DOMAIN Graphics: GPR and Core

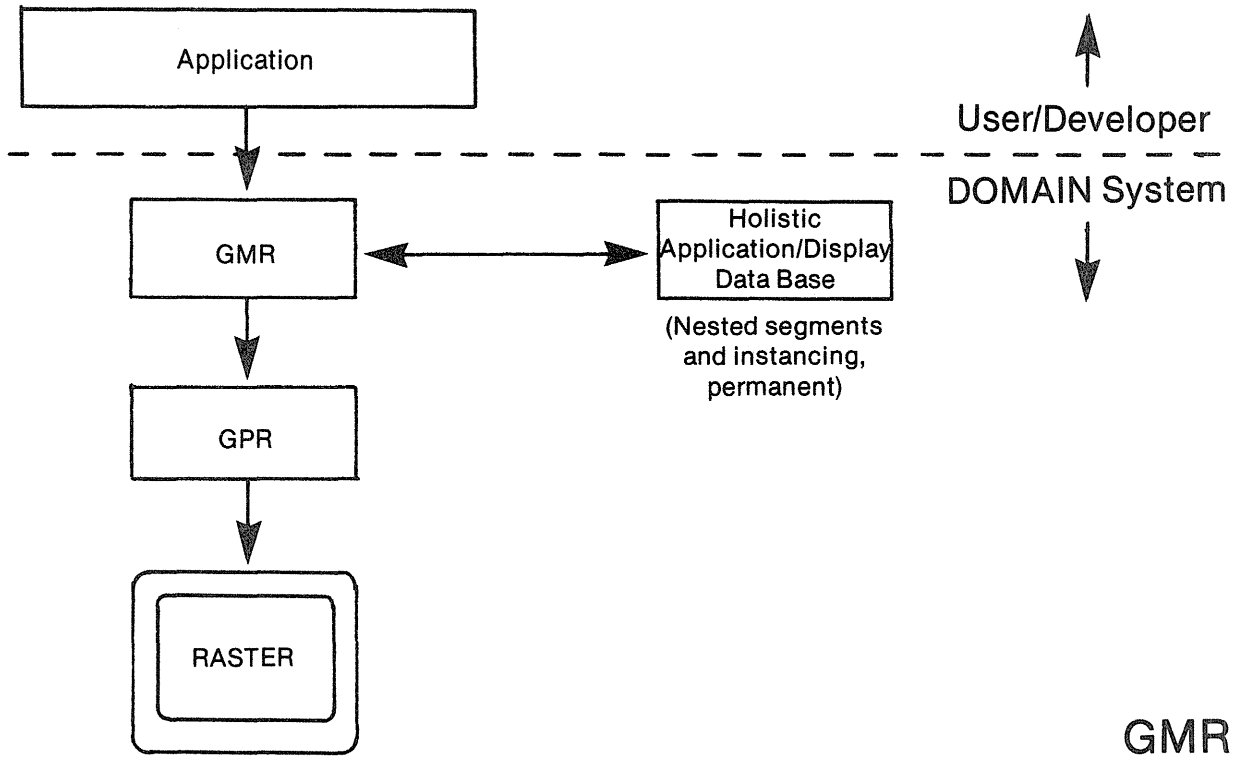

Application
User/Developer
DOMAIN System
GMR
Holistic Application/Display Data Base
(Nested segments and instancing, permanent)
GPR
RASTER
GMR

Figure 1-3. Relationship of DOMAIN Graphics: 2D GMR and GPR

1.6. Processing Model: Viewing Pipeline

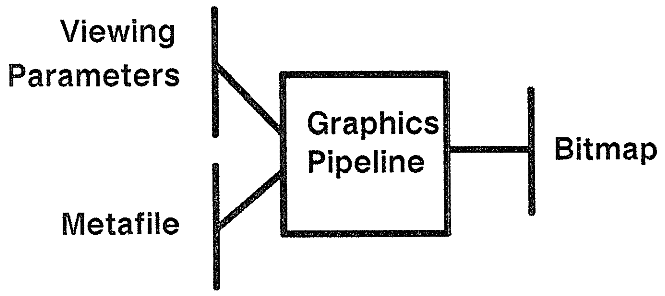


Figure 1-4. Graphics Pipeline

The illustration of the viewing pipeline in Figure 1-4 includes a bitmap. This is a three-dimensional array of bits that can be mapped into one-dimensional address space in several ways:

- Bitmaps in virtual address space may be permanent, residing in the network-wide pathname space.
- Bitmaps may exist in device frame buffers.
- Bitmaps may have associated color lookup tables.

In most applications, the user must perform several operations to display part or all of the graphics data. The graphics package performs most or all of these steps:

- Creates graphics and nongraphics databases.
- Allows modification of databases.
- Allows display of the graphics data including translation, rotation, scaling, and incremental updating of interaction with the display.
- The display process in a graphics application requires reading graphics commands, transforming database coordinates to display coordinates, and displaying graphic entities.

The 2D GMR package performs the steps of the display process as follows:

- Viewing routines call a segment to be displayed in a viewport.

- The display process executes commands in that segment: reads commands, transforms coordinates, and performs display operations.

Instance commands cause commands in another segment to be processed. This can include combining the old transformation and the instance transformation. The display process executes the command in the instanced segment and then restores the old transformation; the process continues executing commands in the instancing segment.

Basic Interactive Processing Model

These are the requirements for the basic graphics processing loop:

- Wait for an input event.

- Change structures or viewing parameters.

- Redisplay the scene through the viewing pipeline.

In a windowing system, these factors can cause complication. Input feedback may have to use the display while the pipeline is running. Several windows may want to share the use of the pipeline.

The viewing pipeline is designed to handle both the processing and the complications. A series of graphics processing instructions flows through the pipeline. A pipeline stage in the process acts on the instruction in one of the following ways:

- Ignores the instruction and passes it on.

- Reads the instruction, updates state (for example, the clipping window), and does not pass the instruction on.

- Transforms the operands and passes the instruction on.

- Converts it to one or more other instructions and passes them on.

The pipeline stages are as follows (not all are relevant to 2D GMR):

- Fetch the instruction.

- Transform the data.

- Clip the view space.

- Execute the projection.

- Clip the screen space.

- Process the drawing.

- Output to the bitmap or display screen.

1.7. Strategies for Developing Applications

This section provides a brief overview of programming strategies. For a more detailed discussion, see Section 15.3.

The 2D GMR package has many features to aid you in developing application packages. The user works in world coordinates that are device-independent. Transformations to images are performed as commands are displayed.

Segmentation includes nested segmentation. This means that instanced segments can themselves be instanced.

The database is optimized for display.

- The command format is optimized for the graphics hardware.
- Segment bounds are stored internally. This allows off-screen segments to be skipped without reading individual commands.
- Special commands, such as the rectangle command, are designed for speed.

2D GMR is in use for a variety of applications. One example is an application that creates a building layout, including floor plans, ducting, and text to describe these parts of the layout. In such a layout, different segments are used for chairs, tables, and desks. These can then be combined by using separate segments for standard rooms and instancing these segments. The floor plan, ducts, and wiring can each have its own segment as well. Text can be contained in a separate segment. Instancing of segments and characteristics such as visibility of segments can change what is displayed.

2D GMR is highly effective for applications that develop printed-wiring boards. This type of application lends itself to the use of separate segments for components and for each collection of connections. When a connection crosses layers, the data can be put into multiple segments.

The approach to applications is to put different classes of data into different segments and to make use of characteristics of segments to change the display. When you are dividing a database into segments, you need to take the following into consideration:

- What elements are repeated.
- What parts of the picture are logically connected. Groups of related segments can be visible or pickable. Segments can be assigned different colors or other attributes when you instance them.
- What is the optimum size for fastest output.

Chapter 2
Displaying Graphic Images

This chapter describes the DOMAIN display, the GM bitmap, and the effect of initialization mode on the display of graphic images. Coordinates systems are defined.

2.1. Elements of the DOMAIN Display

The DOMAIN display is a bit-mapped raster-scan device consisting of these main components: bitmap, display controller, and monitor (see Figure 2-1).

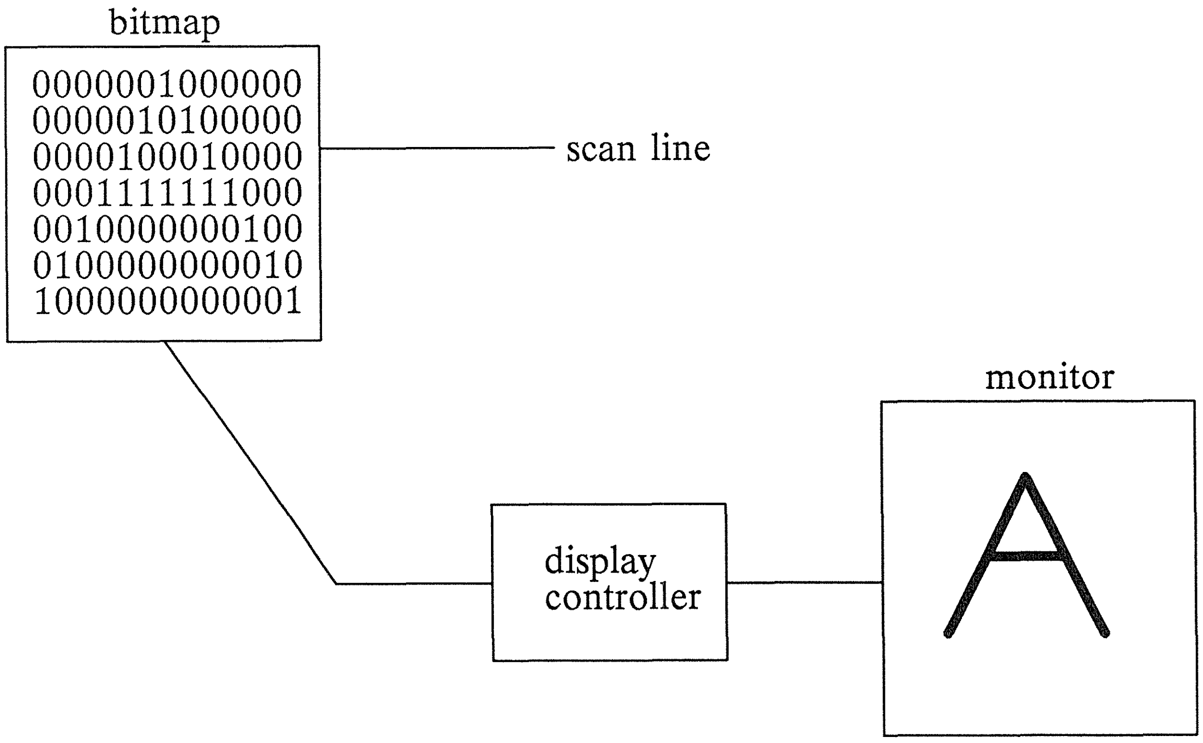


Figure 2-1. A Raster Graphic System

Bitmap

The bitmap (also called a frame buffer) is a data structure used to store values for each point or pixel in a raster. On monochrome displays, there is one bit per pixel. This one-to-one mapping between bits in the bitmap and pixels in the raster has this function: a bit value of 1 turns a pixel on, and a bit value of 0 turns a pixel off. On a color display, more bits are assigned to each pixel to specify color through a color table and pixel values.

GM Bitmap

With the 2D GMR package, you do not have to write data directly to the bitmap. Instead, you use the GM bitmap that is established when you initialize the 2D GMR package. The characteristics of this bitmap depend upon the initialization mode. In direct mode, the GM bitmap is part of the Display Manager window in which the package was initialized. In borrow mode, this is the entire current display. In main-bitmap mode, this is a main-memory bitmap (see Section 3.2).

In 2D GMR, instancing of segments performs the equivalent of bit block transfers and related operations. The 2D GMR package allows you to build your graphics database efficiently and to reuse data with attributes and transformations applied.

Display Controller and Monitor

The display controller is the interface between the bitmap and the display monitor or screen. Its function is to read successive bytes of data from the bitmap and convert this data (0's and 1's) to appropriate video signals, which can then be displayed. The display monitor allows you to view the information you have stored in a bitmap.

2.2. Viewing the Pictures Created by 2D GMR

This section describes the process of displaying the picture data in the metafile. Display modes are explained, along with the following terms particular to the displaying process: GM bitmap, viewport, and view.

GM bitmap is established when you initialize the 2D GMR package. Within the initialization routine, you establish one of five display modes: borrow, direct, main-bitmap, no-bitmap, and within-GPR.

- *Borrow mode*: Uses the entire screen. In borrow mode, the GM bitmap is usually the entire screen.
- *Direct mode*: Displays within a Display Manager window. In direct mode, the GM bitmap is the part of the Display Manager window in which 2D GMR was initialized.
- *Main-bitmap mode*: Displays within a bitmap allocated in main memory. The GM bitmap is this main-memory bitmap.
- *No-bitmap mode*: Allows editing of files without display. There is no GM bitmap.
- *Within-GPR mode*: Displays the output of the metafile within a bitmap that you initialize using routines of the DOMAIN Graphics Primitives package. There is no GM bitmap.

The viewing routines of the 2D GMR package control the form in which metafiles are displayed. When a viewing routine calls for display, the 2D GMR package performs some or all of the commands in the metafile. In all display modes except within-GPR, the picture data is displayed in viewports which are controlled by the 2D GMR package. When you use within-GPR mode, you must specify the exact placement of the picture data within a graphics primitives bitmap under your control. The viewports of the 2D GMR package are not used in this mode.

In the 2D GMR package, a viewport is part or all of the GM bitmap (see Figure 2-2 through Figure 2-6). Each viewport provides a separate view of the output of a metafile or a segment of a metafile. You can see different pictures or parts of pictures in different viewports. Moving the viewport on the GM bitmap does not change the view; the view moves with the viewport.

The view is the part of a picture that is currently seen through a viewport. Moving or scaling a view affects what you see in the viewport (see Figure 2-2 through Figure 2-6). For example, the view may be of a tree. You can move the tree to a new position in the viewport and you can change the size of the tree. The viewport remains the same part of the GM bitmap unless you explicitly change it.

To control the appearance of the view by moving or changing the size of the image, you use viewing transformation routines. These include routines for translating, scaling, and rotating an image in the view.

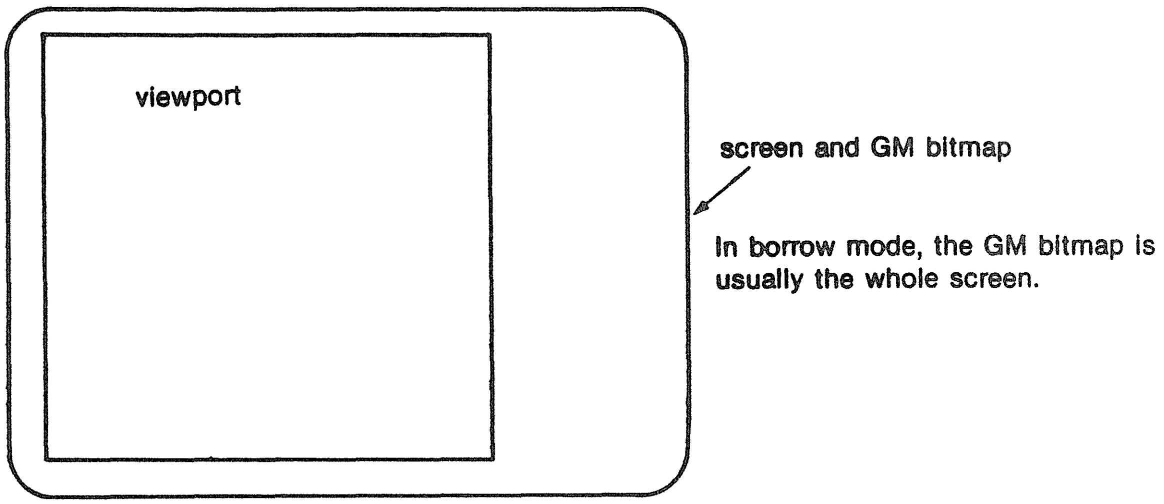


Figure 2-2. Borrow Mode: Screen and GM Bitmap

You can choose to display any segment within a metafile. You can also make other segments referred to (instanced) by this segment be visible or not. Thus, any or all of the segments in a file may be displayed in a particular view. For example, the file may contain a picture with a house, a sign, and trees. You can specify that you want to see the entire file as a view in a viewport. In a different viewport, you may also want to view only the trees, only the sign, or only the house without the sign and trees.

A practical example comes from an architectural application. In developing an architectural design with the 2D GMR package, you may want to display all of a floor plan including details such as ducting and pipes. Alternatively, you may want a less cluttered view showing the floor plan without these details. You can have either one or the other view by choosing the segments you want displayed in the viewport you specify.

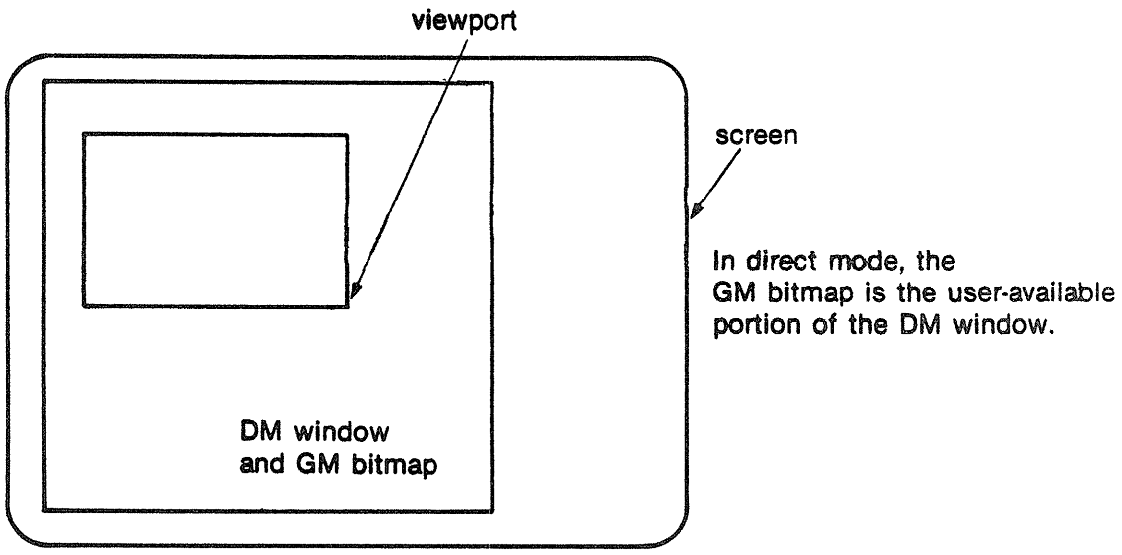


Figure 2-3. Direct Mode: Screen, GM Bitmap, and Viewport

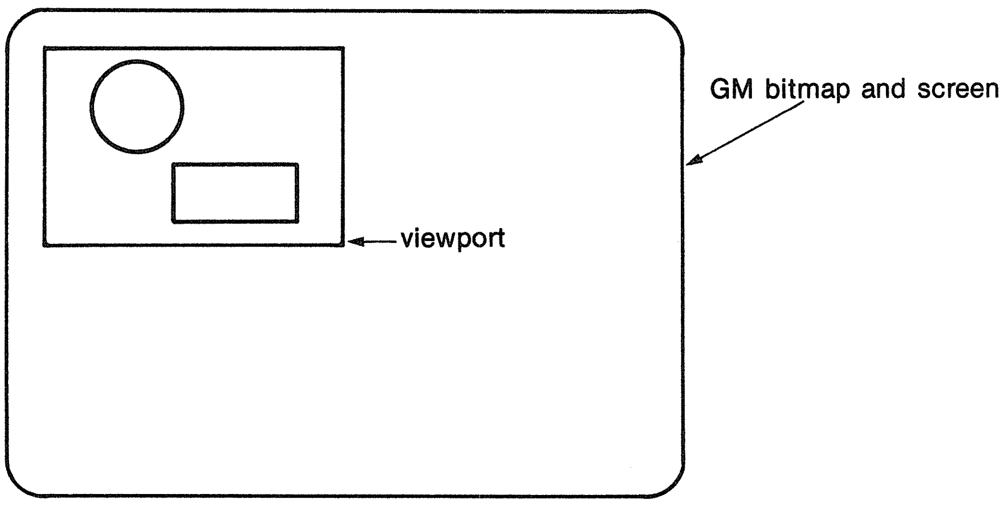


Figure 2-4. Borrow Mode: Viewport

In using the 2D GMR package, you may want to add data while a program is running. You can do this with input routines, which let you generate certain types of data through the keys or buttons on a mouse or puck. This data can be used to calculate parameters for routines that change the appearance of the display (see Chapter 9).

You can use pick routines to select a single entity from a file, either a segment or a command. As you edit the metafile, you can use the pick routines to select the command you want to change. You can also specify that certain elements not be picked. This can protect a basic picture while you change some elements in it (see Section 10.9).

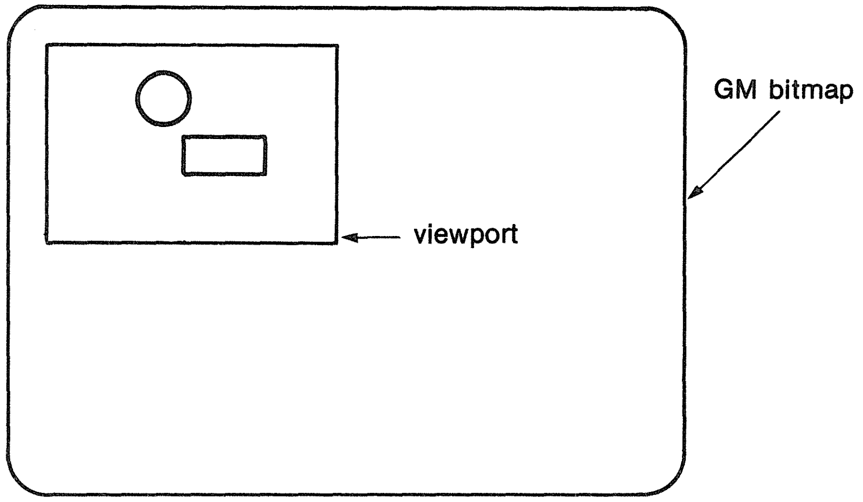


Figure 2-5. Borrow Mode: View Scaled

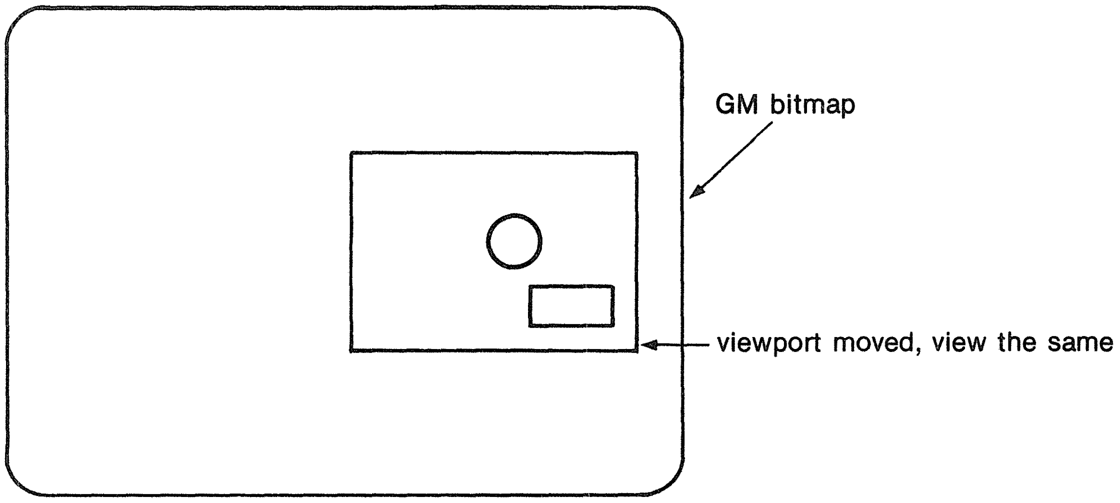


Figure 2-6. Borrow Mode: Viewport and View Moved

2.3. Coordinate Systems

The coordinate system of the 2D GMR package has x increasing to the right and y increasing up. You can develop a picture in terms of the world coordinates that you are accustomed to using for drafting and design (see Figure 2-7).

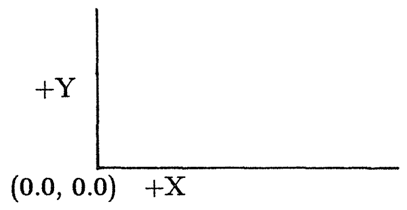


Figure 2-7. 2D GMR Coordinate System

You may use different coordinates in different segments of the picture you develop. You can then specify the relationship of these coordinates when you use instance commands to combine the segments. These different coordinate systems provide flexibility in modeling and displaying graphic images.

In the 2D GMR package, you use world coordinates to create and store a collection of data which generates a picture. The graphics metafile package converts your device-independent world coordinates to device coordinates when it displays the metafile. This allows the same file to be displayed on different DOMAIN nodes without requiring changes in your application program.

This support across devices (device independence) is based on the separation of coordinate systems built into the 2D GMR package. You can use world coordinates to define objects in the two-dimensional world; the package converts these to device coordinates that relate directly to the screen or main-memory bitmap.

Chapter 3
Developing Application Programs

This chapter presents the structure of 2D GMR application programs, including controlling files and segments and instancing segments. The chapter concludes with a basic sample program.

3.1. Structure of 2D GMR Application Programs

The 2D GMR package builds files of picture data stored as collections of 2D GMR commands. Each file of picture data is divided into segments, each of which consists of a sequence of commands (primitive commands, attribute commands, and references to other segments). Every command in a metafile is part of some segment.

The basic structure of a 2D GMR program is as follows:

- Initialize the 2D GMR package.
- Create a file.
- Create a segment within the file.
- Put commands in the segment, for example, to draw a rectangle.
- Display the segment. That is, perform the commands in the segment which make a picture appear on the display.
- Close the metafile.
- Terminate the 2D GMR session.

An application program that uses the routines of the 2D GMR package must first initialize the 2D GMR package. Once the 2D GMR package is initialized, the next step is to create a metafile or to open a previously created one. You must open a file to display or to edit it. You can create or edit segments within this open file; you can insert and delete commands within the segments of the open file.

Once you establish a segment, you may edit and redisplay it. Editing a segment is analogous to editing a line of text with an editor. Every command in a metafile is part of some segment, just as every character in a text file is part of some line.

Segments may contain primitive commands, attribute commands, instance commands, and tag commands.

Primitive commands describe the indivisible, displayable components of a picture, for example, polylines (lists of linked line segments), rectangles, circles, and text.

Attribute commands describe how subsequent components of the picture are to be drawn. For example, one attribute can change the line style from solid to dotted. Another attribute can change the text size. Attribute values may be modified individually or in blocks.

Tag commands are comments in the metafile that do not affect the picture. Tag commands provide a convenient way to track information in the database.

Instance commands cause references to be made to other segments, allowing multiple uses of a single sequence of commands with different transformations applied. Instance commands allow multiple copies of an object to be conveniently drawn in different locations, at different sizes, or with different attributes or color. Instancing of one segment by another segment establishes a hierarchy of segments in the metafile.

Instances can refer to segments which themselves contain instances. This nested segmentation is illustrated in Figure 3-1 and Figure 3-2.

These figures show the structure and display of a file with the following structure. The segment at the top of the hierarchy is "scene." When you display segment "scene," the image is made using the entire contents of the file, that is the complete hierarchy of that file. Segment "scene" instances segment "house." Segment "scene" instances segment "tree" three times. The instances include data for scaling and translation. The result is three trees of different sizes in different locations. Segment "house" instances and translates segment "window" eight times. This results in eight windows at different locations. Segment "house" also instances segment "door" and segment "text." This puts the sign "Grand Motel" on the house. See Appendix D for the program used to create this figure.

The hierarchical structure and instancing speed your development of graphic images by allowing you to do the following:

- Reuse segments by changing transformations.
- View all or part of a metafile.

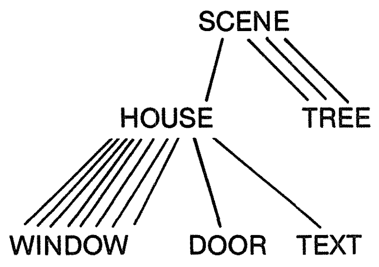


Figure 3-1. Example of Hierarchical Structure

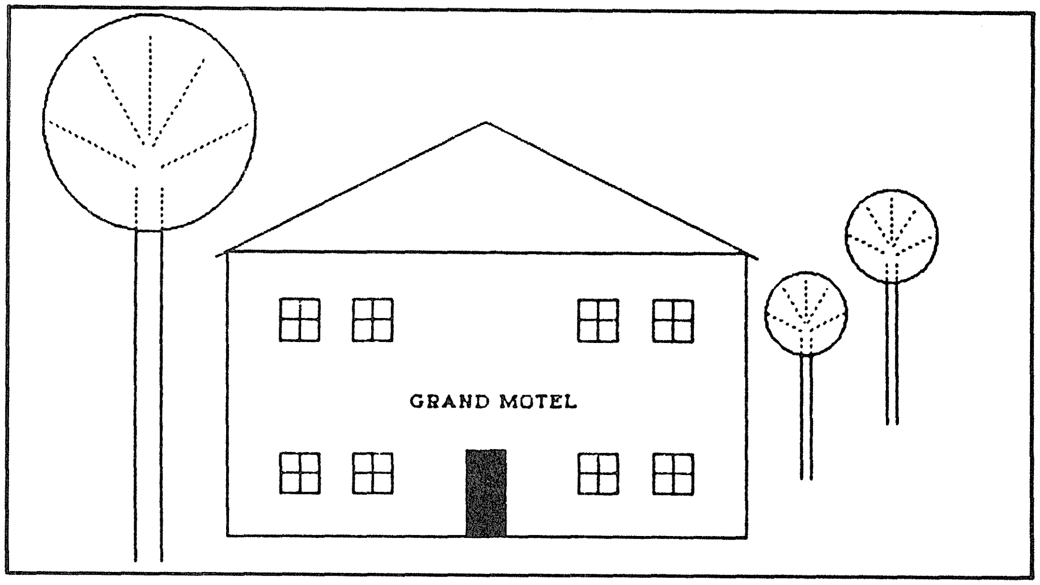


Figure 3-2. Display of File: Hierarchy with Instancing

3.2. Controlling the 2D GMR Package

NOTE: This manual describes the routines of the 2D GMR package in conceptual and procedural terms. For a detailed description of the parameters of these routines, see Volume 1 of the *DOMAIN System Call Reference.*

```
Functions:

GM_$INIT
GM_$TERMINATE
```

To use the 2D GMR package, you must initialize it. At the end of a program that uses 2D GMR, you must terminate the package.

GM_$INIT initializes the 2D GMR package. Within this routine, you establish one of five modes. The choice of mode depends on the purpose of your program and the environment in which you want the program to run. For example, direct mode is desirable if you want the Display Manager environment to be available while this program is running and displaying.

The 2D GMR package does not require that you operate directly on a bitmap (a three-dimensional array of bits having height, width, and depth). Instead when you establish either of two of the five modes (borrow and direct), the 2D GMR package creates a bitmap for display purposes.

The five modes of the 2D GMR package are borrow, direct, main-bitmap, no-bitmap, and within-GPR as shown in Table 3-1.

Table 3-1. **Five Display Modes**

Borrow	On the full screen, which is temporarily borrowed from the Display Manager
Direct	Within a Display Manager window, which is acquired from the Display Manager
Main-bitmap	Using a bitmap allocated in main memory without a display bitmap. This corresponds to no-display mode in the Graphics Primitives package.
No-bitmap	Without a main-memory or display bitmap
Within-GPR	Using a bitmap specified by routines of the DOMAIN Graphics Primitives package

In *borrow mode*, the 2D GMR package borrows the full screen and the keyboard from the Display Manager and uses the display driver directly through 2D GMR software. All windows disappear from the screen. The Display Manager continues to run during this time. However, it does not write the output of any other processes to the screen or read any keyboard input until the 2D GMR package is terminated. Input you have typed ahead into input pads can be read by the related processes while the display is borrowed.

Borrow mode has the advantage of using the entire screen. However, because borrow mode takes over the entire display from the Display Manager, other processes are not immediately available.

Direct mode is similar to borrow mode, but the 2D GMR package borrows a window from the Display Manager instead of borrowing the entire display. The 2D GMR package acquires control of the display each time it must generate graphics output within the borrowed window. All other processes are handled normally by the Display Manager.

Direct mode offers a graphics application the performance and unrestricted use of display capabilities found in borrow mode. In addition, direct mode permits the application to coexist with other activities on the screen. Direct mode is the preferred mode for most interactive graphics applications.

In *main-bitmap mode,* the 2D GMR package creates a main-memory bitmap, but does not create a display bitmap. To display the file on the screen, you must terminate main-bitmap mode and reinitialize in borrow or direct mode.

This mode allows you to create user-available bitmaps larger than the full display.

No-bitmap mode allows you to build a file without a main-memory bitmap or display. No viewing operations may be performed in this mode. To display the file, you must terminate no-bitmap mode and reinitialize in borrow or direct mode.

This mode provides the most efficient way to create a metafile from a data base when you do not need to be simultaneously monitoring a graphic display of the picture.

Within-GPR mode allows you to retain control of the display. You may lay out the display by using the routines of the graphics primitives package. To use this mode, do the following:

- First, initialize the GPR package with GPR_$INIT.
- You may call GPR routines as you wish.
- Next, initialize the 2D GMR package using GM_$INIT and specifying within-GPR mode.
- You may call GPR routines or certain 2D GMR routines. All 2D GMR routines that establish or edit metafiles are available. A certain set of 2D GMR display routines is available (see Section 11.1.2).
- In this mode, 2D GMR displays to the display bitmap that you have established within the graphics primitives package and uses your GPR-specified attribute blocks.

The 2D GMR viewport routines are not available.

GM_$TERMINATE closes the 2D GMR package and closes the display. The package closes any files and segments which have been left open, saving all changes.

3.3. Controlling Files

```
Functions:

GM_$FILE_CREATE
GM_$FILE_OPEN
GM_$FILE_CLOSE
GM_$FILE_SELECT
```

After initializing the 2D GMR package, you must create and open a file using GM_$FILE_CREATE or open an existing file using GM_$FILE_OPEN. This becomes the current file. Within this file, you create segments into which you insert and store commands.

When you use the routine GM_$FILE_CREATE, you give the file a pathname; the package assigns an identification number as an output parameter of the routine. This identification number is an output parameter when you open an existing file using GM_$FILE_OPEN. You use this identification number for reference if you have more than one file open at a time.

To read or edit an existing file, you must open it with GM_$FILE_OPEN.

You may have more than one file open at a time. When you open a file while another file is open, the newly opened file becomes the current file, and the context of the old file (for example, current segment, current command) is saved. You may switch among open files using GM_$FILE_SELECT.

You can perform many normal Shell functions on these files. You can copy (cpf), move (mvf), and delete (dlf) them, but you cannot concatenate (catf) them.

When you close the current file, the package is left with no current file; you must then select a file in order to proceed. Upon completion of editing or using a file, you must close it with GM_$FILE_CLOSE.

3.4. Controlling Segments

Functions:

```
GM_$SEGMENT_CREATE
GM_$SEGMENT_OPEN
GM_$SEGMENT_INQ_ID
GM_$SEGMENT_INQ_CURRENT
GM_$SEGMENT_INQ_NAME
GM_$SEGMENT_INQ_COUNT
GM_$SEGMENT_RENAME
GM_$SEGMENT_CLOSE
GM_$SEGMENT_DELETE
```

The commands within a file are grouped into segments. You must open a segment before you can add commands to it. You can create a new segment with GM_$SEGMENT_CREATE or open an existing segment for redisplay or editing with GM_$SEGMENT_OPEN. This new or newly opened segment becomes the current segment. Only one segment per file may be open at a time.

When you create a segment, you give it a name that must be different from all other segment names in the file. The 2D GMR package assigns the segment an identification number. You can use this returned segment identification number to create references to (instances of) this segment within other segments, or to view this segment (see Section 4.2). The identification number of a segment is stored so that it is retained after you terminate the 2D GMR package.

You also have the option of not naming the segment. To do this, you assign the value 0 to the name length parameter. You then use the segment id number to specify an instance of the segment.

Note that viewing operations are independent of editing operations. A segment need not be open in order to display it.

Use GM_$SEGMENT_CLOSE to close the current segment. You can specify whether or not you want to save the changes you have made.

You can retrieve the name and identification number of the current segment using GM_$SEGMENT_INQ_CURRENT. Use GM_$SEGMENT_INQ_ID to retrieve the identification number of any existing segment in the current file for which you know the segment name. Use GM_$SEGMENT_INQ_NAME to retrieve the name of any existing segment in the current file for which you know the identification number.

You can retrieve the number of segments and maximum segment identification number in a file by using GM_$SEGMENT_INQ_COUNT. This allows you to reopen a file and determine the range of segment identification numbers in the file. You can then obtain a list of segment names using GM_$SEGMENT_INQ_NAME.

You may want to rename a segment before, or during the process of, editing it. To do this, use GM_$SEGMENT_RENAME. You may rename any segment, not just the current segment. You can also assign the value of zero to the name length parameter and then rely on the segment id to identify the segment and to create instances of the segment.

GM_$SEGMENT_DELETE deletes the current segment. You must open a segment before you can delete it. If there are any references to (instances of) this segment in other segments of this file, the segment is not deleted.

The routine GM_$SEGMENT_COPY copies the entire contents of another segment into the current segment. GM_$SEGMENT_COPY is an editing function and is described in more detail in Section 10.11.2.

3.5. Primary Segment

Functions:

```
GM_$FILE_SET_PRIMARY_SEGMENT
GM_$FILE_INQ_PRIMARY_SEGMENT
```

The segments in the metafile have a hierarchical structure. The primary segment can be thought of as the root for the hierarchy of segments in the metafile. As such, the primary segment is assumed to be the start of the picture. When the routine GM_$DISPLAY_FILE is called, the primary segment is displayed.

The first segment you create becomes the primary segment. In Figure 3-1, the primary segment is "scene." When you display "scene," you see the entire picture. The segments are instanced according to the hierarchy established by the primary segment.

Using GM_$FILE_SET_PRIMARY_SEGMENT, you can specify that you want another segment as the primary segment. For example, with "house" as the primary segment in Figure 3-1, you see the following upon display: house, door, eight windows, and sign. You do not see any trees.

If you instance the primary segment from a segment which is not itself instanced, the primary segment is changed to the instancing segment. If you instance this segment from a segment which is itself instanced, the primary segment is changed to the highest-numbered segment not instanced by any other segment.

Use GM_$FILE_SET_PRIMARY_SEGMENT to change the primary segment number. Use GM_$FILE_INQ_PRIMARY_SEGMENT to retrieve the number of the primary segment.

3.6. Using World Coordinates

Coordinate data is supplied to the 2D GMR package as world coordinates. This means that you may define coordinates in the most appropriate form for the application. This flexibility allows for a separate collection of nongraphics data attached to the graphics data.

The coordinate data that you supply is device-independent. The capabilities of the display device are not a matter of concern at the time you are building the file.

When the file is displayed, the transformation from the device-independent coordinates stored in the file to the display coordinates is performed every time the file is displayed as part of the display process. The file coordinates are left alone; the process of viewing does not cause any of the coordinates in the database to be changed.

It is useful to keep in mind the needs of the nongraphics data when you decide on the form of the graphics data. This is up to the application developer who can define whatever coordinates are desired for storage in the metafile. The coordinates are transformed in the metafile to the user-defined coordinates; they are not transformed to some intermediary coordinates first.

With care, you may define different coordinate systems in different segments. For example, some of the segments that are part of the metafile may have coordinates that are millimeters and others that are meters. You must remember, however, at the time you are making reference from one segment to another to perform the action that will convert (coerce) from one coordinate system to the other (see Section 7.3).

3.7. Writing 2D GMR Application Programs

The steps required to produce a 2D GMR application program are presented with a sample program in the sections below.

Including Insert Files

To write 2D GMR application programs, you must include two insert files for the language you are using. The first insert file allows you to use system routines:

FORTRAN	/sys/ins/base.ins.ftn
Pascal	/sys/ins/base.ins.pas
C	/sys/ins/base.ins.c

The second insert file allows you to use 2D GMR routines:

FORTRAN	/sys/ins/gmr.ins.ftn
Pascal	/sys/ins/gmr.ins.pas
C	/sys/ins/gmr.ins.c

Declaring Variables

To use 2D GMR calls, you must declare the variables used as parameters so that they correspond to the data types of the DOMAIN system. For information on data types, see the 2D GMR Data Types section at the beginning of "2D GMR Calls" in the *DOMAIN System Call Reference*, Volume 1.

Initializing the 2D GMR Package

To execute 2D GMR calls in an application program, you must first initialize the package. To do this, call GM_$INIT in the application program.

Preparing an Algorithm to Perform a Task

The next step in the development of a 2D GMR application program is to prepare an algorithm using 2D GMR routines to accomplish the task at hand (for an example, see Section 3.8).

Terminating a 2D GMR Session

To end a 2D GMR session, use GM_$TERMINATE. In terminating the session, this routine closes any open files and saves the changes. The routine also closes any open segments and saves the changes.

3.8. A Program to Draw a Rectangle

The program in this section demonstrates how to initialize the 2D GMR package, create a metafile, create a segment, and draw a rectangle (see Figure 3-3).

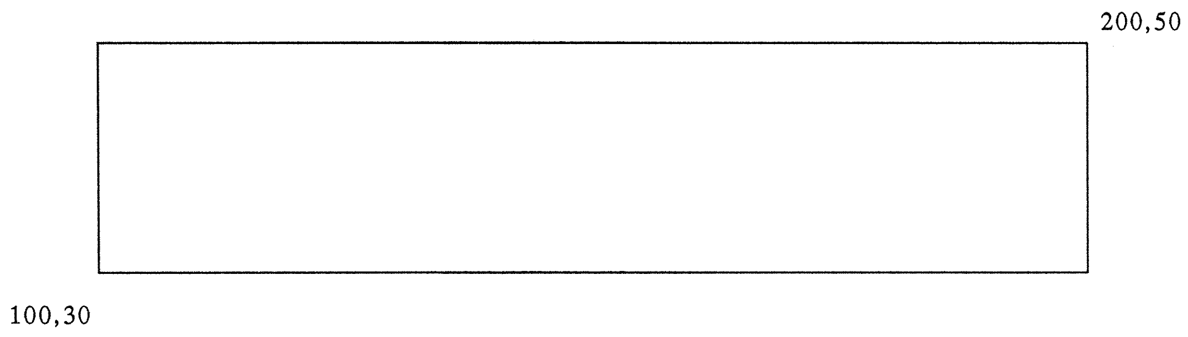


Figure 3-3. Drawing a Rectangle

An additional insert file, /sys/ins/time.ins.pas, is included in this program so that the routine time_$wait is available. This routine is not part of 2D GMR, but is useful to keep a figure displayed on the screen.

GM_$INIT initializes the 2D GMR package in direct mode. Bitmap_size, an input parameter in GM_$INIT, is assigned dimensions of 1024 x 1024. This ensures that the entire window is used as a viewport. When you assign bitmap_size dimensions of 500 x 500 and run the program in a large window, only a portion of the window is used: the top left-most 500 x 500 pixels. The input parameter n_planes is initialized to 8. This is the maximum number of available planes for eight-plane color nodes. On monochrome nodes and four-plane color nodes, this value is interpreted as 1 or 4, respectively.

GM_$FILE_CREATE opens a metafile in the current working directory and makes it current. The metafile is opened in overwrite mode, which deletes the previous version if one existed. The concurrency parameter, gm_$1w, allows anyone to read the file, but only one person to write to the file.

GM_$SEGMENT_CREATE opens a segment within the metafile. The segment is named rectang_seg.

GM_$RECTANGLE_16 draws a rectangle using 16-bit coordinate data. The coordinates of the corners of the rectangle are passed to the routine in two records: pt1 and pt2.

GM_$DISPLAY_FILE displays the contents of the metafile on the screen. This routine will display all segments contained within a metafile. Because the file contains only one segment, this is the only segment displayed. In this case, GM_$DISPLAY_SEGMENT would produce the same results.

GM_$SEGMENT_CLOSE and GM_$FILE_CLOSE close the segment and the file, respectively.

```
program draw_rectangle;
%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gmr.ins.pas';
%include '/sys/ins/time.ins.pas';
%list;

CONST
      one_second   = 250000;
      five_seconds = 5  * one_second;
      ten_seconds  = 10 * one_second;


VAR
      file_id           : integer;
      segment_id        : gm_$segment_id_t;
      st                : status_$t;
      pt1, pt2          : gm_$point16_t; { Array of two 2-byte integers }
      i                 : integer32;
      bitmap_size       : gm_$point16_t := [1024,1024];
      pause             : time_$clock_t;
      high_plane        : integer := 8;

BEGIN
                                           { Define the coordinates of the
      pt1.x := 100;                          rectangle to be drawn. }
      pt1.y := 30;
      pt2.x := 200;
      pt2.y := 50;

      gm_$init(                             { Initialize 2D GMR. }
              gm_$direct
              ,stream_$stdout
              ,bitmap_size
              ,high_plane
              ,st
              );

                                            { Create and name a metafile. }
      gm_$file_create(
              'gmfile'
              ,SIZEOF('gmfile')
              ,gm_$overwrite
              ,gm_$1w
              ,file_id
              ,st
```

```
        );

                                        { Create and name a segment. }
gm_$segment_create
        ('rectang_seg'
        ,sizeof('rectang_seg')
        ,segment_id
        ,st
        );
gm_$rectangle_16                            { Insert the rectangle. }
        (pt1
        ,pt2
        ,false
        ,st
        );

gm_$display_file                            { Display the file. }
        (
        st
        );


pause.low32  := five_seconds;
pause.high16 := 0;
time_$wait(
        time_$relative
        ,pause
        ,st
        );

gm_$segment_close(                          { Close the segment.}
        true
        ,st
        );

gm_$file_close(                            { Close the metafile. }
        true
        ,st
        );

gm_$terminate(                             { Terminate 2D GMR. }
        st
        );

END.
```

Extending the Rectangle Program

Try changing the operation mode in GM_$INIT to gm_$borrow. If you initialize the bitmap with dimensions of 1024 x 1024, the viewport will use the whole display. If the dimensions you provide are smaller, only a portion of the display will be used for the viewport.

Chapter 4
Using Basic Modeling Routines

This chapter describes the draw and fill primitives and explains how to insert them in a segment. The procedure for displaying all or part of the file or segment is described. Instancing and transformation routines are presented with a program to illustrate their use.

4.1. Using Draw and Fill Primitives

```
Functions:

GM_$POLYLINE_2D[16,32,REAL]
GM_$RECTANGLE_[16,32,REAL]
GM_$CIRCLE_[16,32,REAL]
GM_$CURVE_2D[16,32,REAL]
GM_$PRIMITIVE_2D[16,32,REAL]
```

Draw and fill primitives are modeling routines that insert single primitive commands into the current segment of the metafile. When the 2D GMR package reads these commands in the course of displaying a file, they cause something to be drawn. The primitive commands include drawing line segments, rectangles, circles, and curves, and filling areas. Generally, one primitive command is inserted into the metafile each time one of these primitive routines is called. These commands include parameters that describe the object to be drawn.

GM_$POLYLINE_2D[16,32,REAL] routines insert a command to draw a polyline (list of linked line segments). The polyline may be open, closed, or closed and filled. In a closed polyline, the first and last points are connected, forming a polygon.

GM_$RECTANGLE_[16,32,REAL] routines insert a command to draw a rectangle. The routine accepts two diagonally opposite corner points of the rectangle. The rectangle command in the file may fill the area of the rectangle or draw only the outline of it.

GM_$CIRCLE_[16,32,REAL] routines insert a command to draw a circle. The routine accepts the center point and the radius of the circle. The circle command in the file may fill the circle or may draw only the outline of it.

GM_$CURVE_2D[16,32,REAL] routines insert a command to draw a specified curve.

GM_$PRIMITIVE_2D[16,32,REAL] routines insert a command to draw a type of displayed item that you define. You define the following for the command that is placed into the file: a list of points, a list of parameters, and a type number. You connect the type number in the command with a display routine that you define using GM_$PRIMITIVE_DISPLAY_2D (see Section 11.2).

This routine is unlike the other draw and fill primitive routines in that you must write the display routine that displays these primitive commands.

4.2. Displaying Files and Segments

```
Functions:

GM_$DISPLAY_FILE
GM_$DISPLAY_SEGMENT
```

In borrow, direct, and main-bitmap modes, the 2D GMR package produces graphics output in the GM bitmap (the screen, Display Manager window, or main-memory bitmap established when the 2D GMR package was initialized). You can see graphics output or other processes through viewports, which are part or all of the GM bitmap. The view is the picture that you can see in a viewport. Moving or scaling a view moves or scales what you see through the viewport.

When you initialize the 2D GMR package, the command GM_$INIT establishes a single viewport that fills the GM bitmap. You may want to change the size of the viewport or create additional viewports.

You can divide the GM bitmap into multiple, nonoverlapping viewports. You can specify that you want parts of the metafile displayed and moved independently in separate viewports.

In viewing the graphics output of the 2D GMR package, you can use viewing routines to control what is displayed and how it appears. These routines do not affect the contents of the file. You can display all of a file or segment, display part of a file or segment, change attributes associated with the view, or change the color map.

When you display an entire file or segment, the view, or picture, is centered in the viewport. When you display part of a segment or file, you establish the physical bounds of the part that you want displayed. The 2D GMR package then centers and scales that part in the viewport. Files or segments may only be displayed in viewports; space on the display that is outside of viewports is always empty.

You can affect the appearance of the picture by inserting attributes into the file individually to change, for example, the line style from solid to dashed (see Chapter 5). Or you can change attributes more efficiently by inserting an attribute class command into the file. Then, by associating different attribute blocks with the class, you can change the view. You can also display the same file or segment differently in different viewports. To do this, you associate an attribute class with different attribute blocks in different viewports (see Section 13.9).

You can display the entire file or segment. The picture is automatically centered in the current viewport, with a scale calculated so that 95% of the viewport is filled in one dimension and does not overflow the viewport in the other dimension.

GM_$DISPLAY_FILE displays the entire current file in the current viewport. The primary segment of the file is displayed.

GM_$DISPLAY_SEGMENT displays the specified segment, but not the entire file, in the current viewport.

4.3. Displaying Part of a File/Segment

Functions:

```
GM_$DISPLAY_FILE_PART
GM_$DISPLAY_SEGMENT_PART
```

You may want to see only part of a graphic image you have developed. For example, you may want to see only the wheel of a car, not the entire body of the car that you have been modeling.

To get part of an image in a view, you use GM_$DISPLAY_SEGMENT_PART or GM_$DISPLAY_FILE_PART to specify, in segment coordinates, the part of the segment (or file) you want displayed. That part of the segment (or file) is centered in the current viewport with a scale automatically set so that the specified part of the file is displayed as follows: One of the two dimensions fills the viewport, and the other dimension does not overflow the viewport.

This allows you to look at the entire file or segment in one viewport and a smaller part of a file or segment in another viewport. The same file or segment can appear in different viewports simultaneously (see Figure 4-1).

GM_$DISPLAY_FILE_PART displays part of the current file in the current viewport. Bounds are in segment coordinates of the primary segment.

GM_$DISPLAY_SEGMENT_PART displays part of the specified segment in the current viewport.

4.4. Using Instancing

You can use instance routines to insert instance commands into the current segment of a metafile. These commands are references to other segments of the metafile. These references, called instances, provide an economical and efficient way to reuse a set of commands. An instance command includes transformation data (translation, scale, or general two-dimensional transformation). This data relates the coordinate system of the instanced segment to the coordinate system of the instancing segment. This transformation data is collected by the 2D GMR package from the parameters in the instance routine. A segment can contain multiple instances of the same segment, with different attributes and transformation matrices. By interspersing instance commands and attribute commands, you may display different instances with different attributes.

You can define world coordinates as 32-bit and the coordinates of all or some of the segments as 16-bit. In instancing a segment multiple times, you point to the data in that segment more than once. When you instance a segment, for example, the segment with the tree in Figure 3-2, you can scale, rotate, and move it. In addition, you can use attributes to change the line width, line style, background, fill value, or other characteristics of the picture.

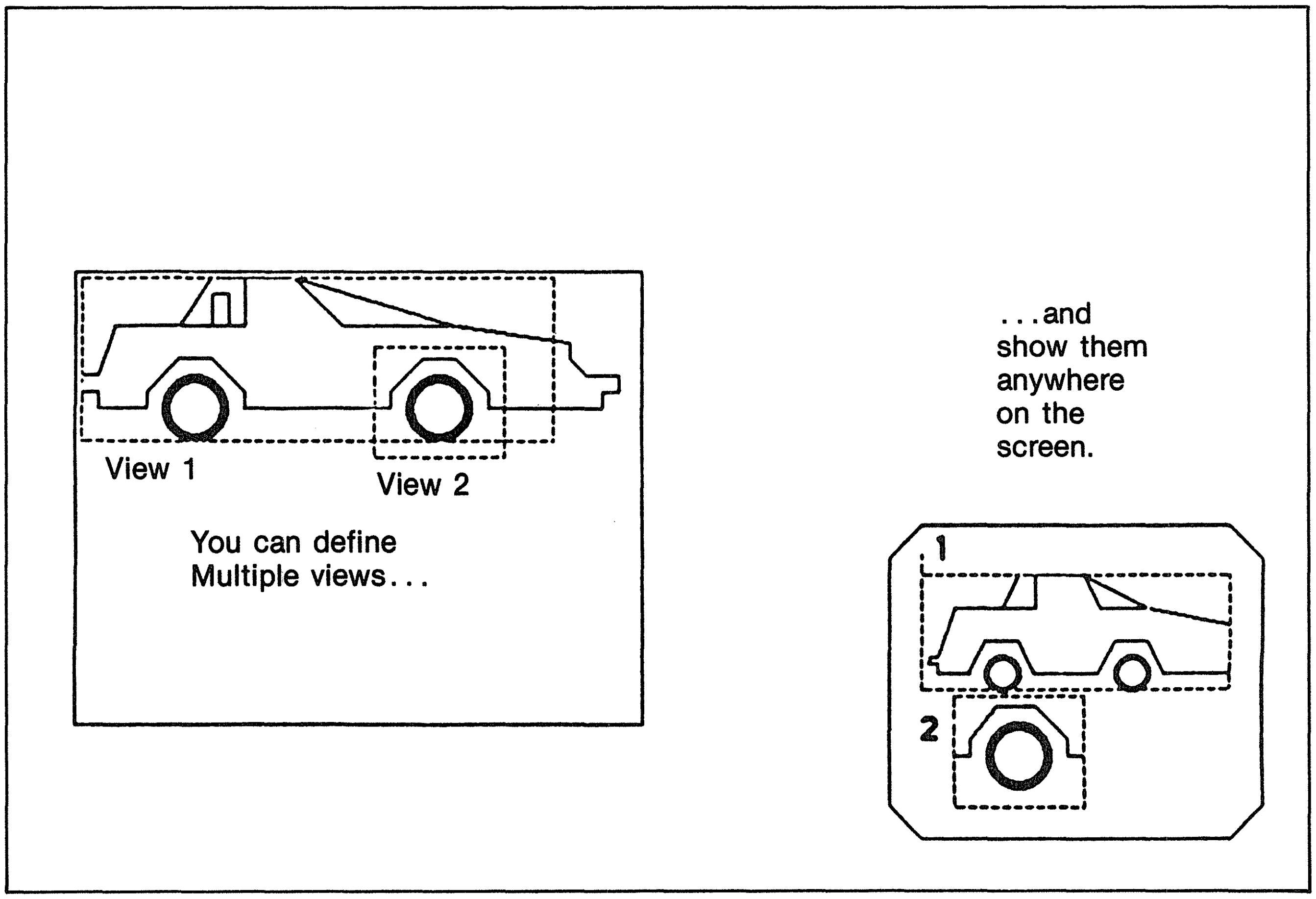

View 1
View 2
You can define Multiple views...
...and show them anywhere on the screen.
1
2

Figure 4-1. Multiple Views Shown in Different Viewports

When you instance a segment using any GM_$INSTANCE... routine, a copy of the instanced segment is not made. Instead, the commands in the instanced segment are displayed, performing the logical equivalent of a subroutine call. When you want to change an instanced segment, you must open that segment and edit it. You cannot edit an instanced segment from an instancing segment.

Instancing of segments may be nested; an instanced segment may contain instances of other segments. However, a segment may not instance itself. An instance may be of any other segment in the file, except that circular instancing is prohibited (instancing a segment which directly or indirectly instances the current segment). For example, if segment "house" contains an instance of segment "door," then you may not insert an instance of segment "house" into segment "door."

A segment must exist before you can instance it.

4.5. Using Transformations

Functions:

```
GM_$INSTANCE_TRANSLATE_2D[16,32,REAL]
GM_$INSTANCE_SCALE_2D[16,32,REAL]
```

GM_$INSTANCE_TRANSLATE_2D[16,32,REAL] routines insert a reference to a segment's identification into the current segment. You must give the (x,y) coordinates of the translation to apply to the referenced segment. This reference is unscaled and unrotated.

GM_$INSTANCE_SCALE_2D[16,32,REAL] routines insert a reference to a segment's identification into the current segment. You must give the (x,y) coordinates of the translation and the scale to apply to the referenced segment. When the command is processed, scaling is performed before translation. This reference is unrotated.

Note that point (0,0) in the coordinates of the instanced segment remains stationary through scaling. Therefore, segments that will be transformed in this way should be centered around (0,0).

When displayed, the segment is scaled by the given scale factor, then translated by the amount (x,y) in segment coordinates of the instancing segment.

4.6. A Program Using Primitives and Instancing

The program in this section draws the design in Figure 4-2.

The program uses two segments: small_rec and large_rec. The segment small_rec draws a small, filled rectangle with dimensions of 100 x 100. The segment large_rec draws a large, unfilled rectangle with dimensions of 500 x 500, instances the segment small_rec four times, and draws two polylines connecting the four filled rectangles.

The routine GM_$INSTANCE_TRANSLATE copies the contents of the instanced segment (in this case, small_rec) and translates it to the position provided. This routine is used four times to produce four rectangles.

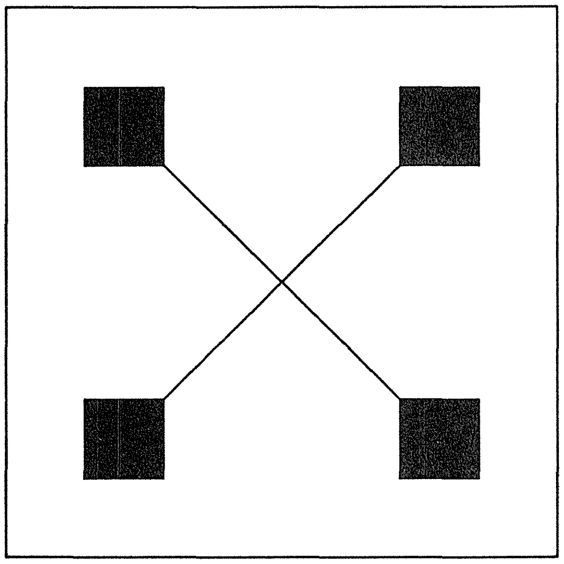


Figure 4-2. Four Filled Rectangles within a Box

The routine GM_$POLYLINE draws a polyline between the end points provided.

The routine GM_$DISPLAY_SEGMENT displays the segment large_rec. When this segment is displayed, the four instances of small_rec are automatically displayed because they have been instanced within large_rec.

```
program draw_rectangle;
%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gmr.ins.pas';
%include '/sys/ins/time.ins.pas';
%list;

CONST
    one_second = 250000;
    five_seconds = 5 * one_second;
    ten_seconds = 10 * one_second;

VAR
      file_id          : integer;
      small_id         : gm_$segment_id_t;
      large_id         : gm_$segment_id_t;
      st               : status_$t;
      pt1, pt2         : gm_$point16_t;
      i                : integer32;
      bitmap_size      : gm_$point16_t := [1024,1024];
      position         : gm_$point16_t;
      positions        : gm_$point_array16_t;
      pattern          : gm_$draw_pattern_t;
      pause            : time_$clock_t;

BEGIN

      gm_$init                                  { Initialize 2D GMR. }
              (gm_$direct
        ,stream_$stdout
        ,bitmap_size
        ,8
        ,st
        );
                                   { Create and name a metafile. }
gm_$file_create
        ('gmfile'
        ,6
        ,gm_$overwrite
        ,gm_$1w
        ,file_id
        ,st
        );

pt1.x := 100;
pt1.y := 100;
pt2.x := 200;
pt2.y := 200;

                                    { Create and name a segment. }
gm_$segment_create
        ('small_rec'
        ,sizeof('small_rec')
        ,small_id
        ,st
        );

gm_$rectangle_16                             { Draw a rectangle. }
        (pt1
        ,pt2
        ,true
        ,st
        );

gm_$segment_close                           { Close the segment. }
        (true
        ,st
        );

                         { Define the coordinates of the rectangle }
pt1.x := 100;            { to be drawn. }

pt1.y := 100;
pt2.x := 600;
pt2.y := 600;


gm_$segment_create
        ('large_rec'
        ,sizeof('large_rec')
        ,large_id
        ,st
        );

gm_$draw_style
        (gm_$solid
        ,4
        ,pattern
        ,0
        ,st
        );


gm_$rectangle_16                                 { Draw a rectangle. }
        (pt1
        ,pt2
        ,false
        ,st
        );                              { Instance the small rectangle }
                                        { four times. }


position.x := 100;
position.y := 100;

gm_$instance_translate_2d16
        (small_id
        ,position
        ,st
        );

position.x := 300;
position.y := 300;

gm_$instance_translate_2d16
        (small_id
        ,position
        ,st
        );

position.x := 300;
position.y := 100;

gm_$instance_translate_2d16
        (small_id
        ,position
        ,st
        );

position.x := 100;
position.y := 300;

gm_$instance_translate_2d16
        (small_id
        ,position
        ,st
        );                             { Draw two polylines connecting }
                                       { the four rectangles. }

positions[1].x := 300;
positions[1].y := 300;
positions[2].x := 400;
positions[2].y := 400;


gm_$polyline_2d16
        (2
            ,positions
            ,false
            ,false
            ,st
            );

    positions[1].x  := 300;
    positions[1].y  := 400;
    positions[2].x  := 400;
    positions[2].y  := 300;


    gm_$polyline_2d16
            (2
            ,positions
            ,false
            ,false
            ,st
            );

    gm_$segment_close                          { Close the segment. }
            (true
            ,st
            );

    gm_$display_segment
            (large_id
            ,st
            );


                                       { Keep figure displayed on the }
    pause.low32  := five_seconds;      { screen for five seconds. }

    pause.high16 := 0;
    time_$wait( time_$relative, pause, st );


    gm_$file_close                             { Close the metafile. }
            (true
            ,st
            );

    gm_$terminate                               { Terminate 2D GMR. }
            (
            st
            );

END.
```

4.7. Instances with Arbitrary Transformations

`Function:`

`GM_$INSTANCE_TRANSFORM_2D[16,32,REAL]`

GM_$INSTANCE_TRANSFORM_2D[16,32,REAL] routines insert a reference to a segment's identification into the current segment. Within this routine, you must specify a general 2 by 2 transformation matrix (that is, rotation, scale, reflection, and skewing) and give the (x,y) coordinates of the translation to apply to the referenced segment. When the command is processed, the 2 by 2 matrix is applied before the translation.

A sample transformation matrix follows:

The input to this routine is a 2x2 transformation matrix, which has the following form:

```
| XX    XY |        | Sx * cos(A)     Sy * sin(A)  |           Sx = Scale in
|          |        |                              |                X direction
|          | ===>   |                              |  where:   Sy = Scale in
|          |        |                              |                Y direction
| YX    YY |        | Sx *-sin(A)     Sy * cos(A)  |            A = Angle of
                                                                    rotation
```

The point (0,0) in the coordinates of the instanced segment remains stationary through reflection, rotation, and scaling. Therefore, you should center segments that will be transformed in these ways around (0,0).

When displayed, the segment is rotated and scaled by the 2x2 matrix. The segment is then scaled by the given scale factor and then translated by the amount (x,y) in segment coordinates of the instancing segment.

4.8. A Technique Using Arbitrary Transformations

In the program hotel.pas in Appendix D, the segment "house" is instanced into the segment "scene" by using GM_$INSTANCE_TRANSLATE:

```
 p[ 1 ].x := 0;
 p[ 1 ].y := 0;

GM_$INSTANCE_TRANSLATE_2D16
    ( sid_house
    , p[ 1 ]
    , status
    );
```

GM_$INSTANCE_TRANSLATE allows you to instance a segment without changing the segment's scale or orientation. GM_$INSTANCE_SCALE allows you to change the scale as well. (This is used for the trees in the same example program.) Even more general transformations are possible with GM_$INSTANCE_TRANSFORM.

The following program fragment replaces the GM_$INSTANCE_TRANSLATE command (used to instance the segment "house" into the segment "scene") with a GM_$INSTANCE_TRANSFORM command. Note the extra argument:

```
p[ 1 ].x := 0;
p[ 1 ].y := 0;

gm_$instance_transform_2d16
    ( sid_house
    , matrix
    , p[ 1 ]
    , status
    );
```

The argument "matrix" is declared as "gm_$rotate_real2x2_t." (The name of this type is misleading in that the matrix may be any 2x2 matrix, not just a rotation matrix. The only restriction is that the matrix must have a nonzero determinant.) Some examples are given here for reference. Try substituting them into the example program hotel.pas in Appendix D.

To translate only, set the matrix (to the identity matrix) as follows:

```
matrix.xx := 1;
matrix.xy := 0;
matrix.yx := 0;
matrix.yy := 1;
```

To translate and scale only, set the matrix as follows:

```
matrix.xx := scale;
matrix.xy := 0;
matrix.yx := 0;
matrix.yy := scale;
```

To rotate the house counterclockwise through an angle of 1 radian (about 57 degrees), set the matrix as follows:

```
matrix.xx :=   COS( 1 );
matrix.xy := - SIN( 1 );
matrix.yx :=   SIN( 1 );
matrix.yy :=   COS( 1 );
```

To rotate the house counterclockwise through an angle of 1 radian and scale uniformly, set the matrix as follows:

```
matrix.xx :=   COS( 1 ) * scale;
matrix.xy := - SIN( 1 ) * scale;
matrix.yx :=   SIN( 1 ) * scale;
matrix.yy :=   COS( 1 ) * scale;
```

To rotate the house counterclockwise through an angle of 1 radian, and scale independently in the x and y directions, set the matrix as follows:

```
matrix.xx :=   COS( 1 ) * scale_x;
matrix.xy := - SIN( 1 ) * scale_y;
matrix.yx :=   SIN( 1 ) * scale_x;
matrix.yy :=   COS( 1 ) * scale_y;
```

To skew the house, as if there were a strong wind blowing from left to right, you might set the matrix as follows:

```
matrix.xx :=  1;
matrix.xy :=  1;
matrix.yx :=  0;
matrix.yy :=  1;
```

Chapter 5
Using Attributes

This chapter describes the use of individual attribute commands, explains how to use attributes in relation to instancing, and provides a program to illustrate these functions.

5.1. Using Draw and Fill Attributes

A metafile can contain attribute commands to change individual attributes. These attributes determine such characteristics as the style and pixel value used in drawing lines and filling areas. The plane mask attribute allows you to specify which planes of a bitmap can be modified by any graphics operation and which planes are protected from modification.

Each of the routines described in this section inserts a command into the current segment to change one attribute. When the segment is displayed, all subsequent primitive commands in the segment are displayed with this new value of the changed attribute. This new attribute value also applies to the segments subsequently instanced from this segment, but never to the segment that instanced this segment (see the examples in Section 5.3).

Attribute commands inserted into a segment of the metafile take precedence over any other means of changing attributes. See Chapter 13 for descriptions of other ways of changing attributes.

The default attribute settings are shown in Table 5-1.

Table 5-1. Default Attribute Settings

ATTRIBUTE	DEFAULT VALUE
Draw Style	Solid line
Draw Value	1
Fill Value	1
Fill Background Value	-2 (same as viewport background)
Fill Pattern	All 1's
Text Value	1
Text Background Value	-2 (same as viewport background)
Text Size	10.0
Font Family ID Number	1
Plane Mask	All planes can be modified
Draw Raster Op	3 (set all destination bit values to source bit values)

5.1.1. Line Attributes

Functions:

```
GM_$DRAW_VALUE
GM_$DRAW_STYLE
```

Line attributes determine the pixel value, style, and width of lines used in modeling commands.

GM_$DRAW_VALUE inserts a command to set the pixel value used when lines are drawn.

GM_$DRAW_STYLE inserts a command to set the line style used to display unfilled polylines and rectangles. Line style can be either solid or a specified pattern.

5.1.2. Fill Attributes

Functions:

```
GM_$FILL_VALUE
GM_$FILL_BACKGROUND_VALUE
GM_$FILL_PATTERN
```

Fill attributes determine the appearance of filled areas.

GM_$FILL_VALUE inserts a command to set the pixel value used when filling an area.

GM_$FILL_BACKGROUND_VALUE inserts a command to set the pixel value used in unfilled parts of a pattern that fills an area. For example, in a checkerboard pattern with only alternate squares filled, this attribute sets the appearance of the unfilled squares.

GM_$FILL_PATTERN inserts a command to set the pattern used to fill the interior of filled areas. The default fill pattern is all 1's, indicating that the fill value is to be used for all pixels in the area being filled. With any other pattern of 1's and 0's, the fill background value is used for pixels corresponding to 0's in the pattern.

5.2. Using Color Map Attributes

Function:

```
GM_$PLANE_MASK
```

GM_$PLANE_MASK specifies which planes of a bitmap can be modified by any graphics operation and which planes are protected from modification.

Color map operations are described in Section 14.1.

5.2.1. Raster Operation Attributes

Function:

GM_$DRAW_RASTER_OP

Raster operation attributes allow you to specify two conditions that determine what appears in the bitmap:

- What you are drawing.

- What was there in the GM bitmap.

There are sixteen different rules for combining old values and values being drawn to create new values. A different raster operation code exists for each of these sixteen rules (see Section 5-2).

A raster operation specifies how to combine source pixel values and destination pixel values to form new destination values. The source values are determined by GM_$DRAW_VALUE. The value of each new destination bit is assigned by a Boolean function of the previous value of each destination bit and the value of the corresponding source bit.

Sixteen raster operations form the set of rules for combining bit values. Assigning a raster operation code alters no values. The raster operation code controls how values are logically combined when a program subsequently draws, fills, or writes text. Table 5-2 lists the op codes, symbolic constants, and logical functions for the sixteen raster operations. The symbolic constants are available in the insert files for Pascal, FORTRAN, and C. Table 5-3 is a truth table of the raster operations.

GM_$DRAW_RASTER_OP inserts a command to set the logical raster operation to be performed when any nonfilled primitive is drawn.

Table 5-2. Raster Operations and Their Functions

OP CODE	CONSTANT	LOGICAL FUNCTION
0	GM_$ROP_ZEROS	Assign 0 to all new destination values.
1	GM_$ROP_SRC_AND_DST	Assign source AND destination to new destination.
2	GM_$ROP_SRC_AND_NOT_DST	Assign source AND complement of destination to new destination.
3	GM_$ROP_SRC	Assign all source values to new destination. (Default)
4	GM_$ROP_NOT_SRC_AND_DST	Assign complement of source AND destination to new destination.
5	GM_$ROP_DST	Assign all destination values to new destination.
6	GM_$ROP_SRC_XOR_DST	Assign source EXCLUSIVE OR destination to new destination.
7	GM_$ROP_SRC_OR_DST	Assign source OR destination to new destination.
8	GM_$ROP_NOT_SRC_AND_NOT_DST	Assign complement of source AND complement of destination to new destination.
9	GM_$ROP_SRC_EQUIV_DST	Assign source EQUIVALENCE destination to new destination.
10	GM_$ROP_NOT_DST	Assign complement of destination to new destination.
11	GM_$ROP_SRC_OR_NOT_DST	Assign source OR complement of destination to new destination.
12	GM_$ROP_NOT_SRC	Assign complement of source to new destination.
13	GM_$ROP_NOT_SRC_OR_DST	Assign complement of source OR destination to new destination.
14	GM_$ROP_NOT_SRC_OR_NOT_DST	Assign complement of source OR complement of destination to new destination.
15	GM_$ROP_ONES	Assign 1 to all new destination values.

Table 5-3. Raster Operations: Truth Table

SOURCE BIT VALUE	DESTINATION BIT VALUE	RESULTANT BIT VALUES FOR THE FOLLOWING OP CODES: 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

5.3. Using Attributes and Instancing

Attribute commands change the attributes for all subsequent primitive commands in the segment. These changed attributes also apply to all primitive commands in segments subsequently instanced from this segment. If an attribute command occurs in a segment that is instanced from another segment, it only affects the subsequent commands in instanced segment, never the instancing segment. The program fragment below provides an example.

```
The following sequence of routines sets up two segments:

gm_$segment_create('bottom',6,bottomid,status);
gm_$rectangle_16(point1,point2,false,status);
gm_$draw_value(2,status);
gm_$circle_16(center,radius,false,status);
gm_$segment_close(true,status);

gm_$segment_create('top',3,topid,status);
gm_$draw_value(4,status);
gm_$instance_translate_2d16(bottomid,translate2,status);
gm_$rectangle_16(point3,point4,false,status);
gm_$segment_close(true,status);

These two segments will then contain the following commands:

           'TOP':  DRAW VALUE (4)
                   INSTANCE  ('BOTTOM')
                   RECTANGLE (POINT3,POINT4,DONT_FILL)

        'BOTTOM':  RECTANGLE (POINT1,POINT2,DONT_FILL)
                   DRAW VALUE (2)
                   CIRCLE (CENTER,RADIUS,DONT_FILL)
```

When a viewing routine displays segment 'TOP', it does the following:

- Draws the rectangle (point1,point2) using draw value 4, since that attribute was set by the instancing segment and has not been changed.

- Draws the circle using draw value 2, the most recent value assigned in this segment.

- Draws the rectangle (point3,point4) using draw value 4, since attribute values changed by the instanced segment are restored to their previous values before returning control to the instancing segment.

5.4. A Program with Attributes and Instancing

The program presented in this section modifies the program presented in Section 4.6.

The program in this section uses two segments: small_rec and large_rec. The segment small_rec draws a small filled rectangle with dimensions of 100 x 100. The segment large_rec draws a large unfilled rectangle with dimensions of 500 x 500, instances the segment small_rec four times, and draws two polylines connecting the four filled rectangles.

The routine GM_$INSTANCE_TRANSLATE copies the contents of the instanced segment (in this case, small_rec) and translates it to the position provided. This routine is used four times to produce four rectangles.

The routine GM_$POLYLINE draws a polyline between the endpoints provided.

The routine GM_$DISPLAY_SEGMENT displays the segment large_rec. When this segment is displayed, the four instances of small_rec are automatically displayed because they have been instanced within large_rec.

The program presented in this section changes the program presented in Section 4.6 as described below.

In this program, the rectangle drawn by the segment small_rec is no longer filled, and the value of the line drawing attribute is changed. In the segment large_rec the line drawing attribute is changed as well.

In segment small_rec, the routine GM_$DRAW_STYLE changes the line style to gm_$dotted. This is the simplest way to produce dotted or dashed lines, provided the style is acceptable. The parameter gm_$dotted causes every other pixel in the line to be illuminated.

In segment large_rec, GM_$DRAW_STYLE also changes the line style. This time the style is defined so that each dash is twelve pixels long, and each space between the dashes is four pixels long. The line style is defined in the array "pattern" (see the variable declaration section of the program). This is an eight-element character array. Char(2#11111111) sets eight bits on, and char(2#11110000) sets four bits on and four off. Only sixteen bits of this array are used to define the line-style.

The line attribute used in small_rec (the instanced segment) is not affected by the line attribute in large_rec (the instancing segment).

```
program draw_rectangle;
%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gmr.ins.pas';
%include '/sys/ins/time.ins.pas';
%list;
```

```
CONST

    one_second = 250000;
    five_seconds = 5 * one_second;


VAR
      file_id          : integer;
      small_id         : gm_$segment_id_t;
      large_id         : gm_$segment_id_t;
      st               : status_$t;
      pt1, pt2         : gm_$point16_t;
      i                : integer32;
      bitmap_size      : gm_$point16_t := [1024,1024];
      position         : gm_$point16_t;
      positions        : gm_$point_array16_t;
      pattern          : gm_$draw_pattern_t;
      pause            : time_$clock_t;
BEGIN

      gm_$init                       { Initialize 2D GMR. }
              (gm_$direct
              ,stream_$stdout
              ,bitmap_size
              ,8
              ,st
              );
                                   { Create and name a metafile. }
      gm_$file_create
              ('gmfile'
              ,6
              ,gm_$overwrite
              ,gm_$1w
              ,file_id
              ,st
              );

      pt1.x := 100;
      pt1.y := 100;
      pt2.x := 200;
      pt2.y := 200;

                                     { Create and name a segment.' }
      gm_$segment_create
              ('small_rec'
              ,sizeof('small_rec')
              ,small_id
              ,st
              );

      gm_$rectangle_16               { Draw a rectangle. }
              (pt1
              ,pt2
              ,true
              ,st
              );

      gm_$segment_close              { Close the segment. }
              (true
          ,st
          );


pt1.x := 100;
pt1.y := 100;
pt2.x := 600;
pt2.y := 600;


gm_$segment_create
          ('large_rec'
          ,sizeof('large_rec')
          ,large_id
          ,st
          );

gm_$draw_style
          (gm_$solid
          ,4
          ,pattern
          ,0
          ,st
          );


gm_$rectangle_16           { Draw an unfilled rectangle. }
          (pt1
          ,pt2
          ,false
          ,st
          );

position.x := 100;
position.y := 100;

gm_$instance_translate_2d16   { Instance and move the small rectangle. }
          (small_id
          ,position
          ,st
          );

position.x := 300;
position.y := 300;

gm_$instance_translate_2d16
          (small_id
          ,position
          ,st
          );

position.x := 300;
position.y := 100;

gm_$instance_translate_2d16
          (small_id
          ,position
          ,st
          );
```

```
 position.x := 100;
 position.y := 300;

 gm_$instance_translate_2d16
         (small_id
         ,position
         ,st
         );

 positions[1].x  := 300;
 positions[1].y  := 300;
 positions[2].x  := 400;
 positions[2].y  := 400;


 gm_$polyline_2d16
         (2
         ,positions
         ,false
         ,false
         ,st
         );

 positions[1].x  := 300;
 positions[1].y  := 400;
 positions[2].x  := 400;
 positions[2].y  := 300;


 gm_$polyline_2d16
         (2
         ,positions
         ,false
         ,false
         ,st
         );

 gm_$segment_close                { Close the segment. }
         (true
         ,st
         );

 gm_$display_segment
         (large_id
         ,st
         );


                                 { Display the figure for five seconds. }
pause.low32  := five_seconds;
pause.high16 := 0;
time_$wait
         ( time_$relative
         , pause
         , st
         );


gm_$file_close                    { Close the metafile. }
```

```
            (true
            ,st
            );

    gm_$terminate                        { Terminate 2D GMR. }
            (
            st
            );

END.
```

Chapter 6
Using Modeling Routines: Text

This chapter explains how to insert text and text attributes into a segment. Font families and their use are described along with techniques for creating stroke fonts. Programs to illustrate the routines are included.

6.1. Using Text

A modeling routine of the 2D GMR package inserts text into the file. The text string with its size is inserted into the current segment.

A font is a related set of characters used for text. In programming terms, a font is data that graphically describes a set of related character images. These images may be developed by specifying the pixels that make up each character in a bitmap (pixel fonts), or by specifying the end points of vectors that make up each character (stroke fonts). Both pixel and stroke fonts can be stored in named files on a node. Discussion of both types is included in this chapter.

Families of text fonts provide a convenient way to display text in different sizes as the displayed picture grows and shrinks. A font family is a group of fonts of the same style with a range of sizes. During display operations, the 2D GMR package selects a font of the appropriate size from the font family.

6.2. Inserting Text

Functions:

GM_$TEXT_2D[16,32,REAL]

GM_$TEXT_2D[16,32,REAL] inserts a text string into the current segment. The routine includes specification of the length and the starting point of the string, in segment coordinates.

The text is placed as follows: The first character of the text string is placed at the location you specify. This means that the origin of this character, as defined in the font, is placed at the specified location. Usually, the origin is the lower left-hand corner, excluding descenders.

You may also specify the direction (in degrees) in which text is to be written. A value of 0.0 indicates left to right. Other values indicate clockwise rotation. For example, -90 degrees indicates bottom to top.

You may not insert text until you have included at least one font family in the metafile (see Section 6.4).

6.3. Using Text Attributes

Functions:

```
GM_$TEXT_VALUE
GM_$TEXT_BACKGROUND_VALUE
GM_$TEXT_SIZE
GM_$FONT_FAMILY
```

A program can set text attributes. These attributes determine the pixel value, font family, and size of text. Attributes can be set individually or in other ways. For a list of default attribute settings, see Table 5-1. For a discussion of attribute classes and attributes in viewing operations, see Chapter 13.

GM_$TEXT_VALUE specifies the pixel value to be used in writing text. GM_$TEXT_BACKGROUND_VALUE changes the background pixel value to be used in writing text. These attributes are for pixel fonts only. Stroke fonts use GM_$DRAW_VALUE to establish pixel values.

GM_$TEXT_SIZE specifies the maximum height of a character from the the font family you have specified.

GM_$FONT_FAMILY specifies the font family to use in writing text. Font families are explained below.

6.4. Identifying Font Families

Functions:

```
GM_$FONT_FAMILY_INCLUDE
GM_$FONT_FAMILY_INQ_ID
GM_$FONT_FAMILY_RENAME
GM_$FONT_FAMILY_EXCLUDE
```

A font family is a collection of fonts, each of a different size. The file names of the fonts are listed in a font family file. This is an ASCII file, listing file names of fonts, one font per line. You may build your own font family files that list names of pixel font files or stroke font files.

In a font family file, lines which start with "#" are treated as comments and ignored. Currently, fonts must be listed in order of size from largest to smallest. For example, a font family file can include these lines:

```
/sys/dm/fonts/f9x15
/sys/dm/fonts/f7x13
/sys/dm/fonts/f5x9
/sys/dm/fonts/f5x7
```

During a display operation, the 2D GMR package selects a font of the appropriate size from the font family.

To use a font family in writing text, use GM_$FONT_FAMILY_INCLUDE. You specify the font family's pathname and pathname length, and the type of font (pixel or stroke). All fonts within the font family file must be of the specified type. The 2D GMR package returns the font family identification number. You can then use this identification number in referencing the font family. Use GM_$FONT_FAMILY_INQ_ID to retrieve the font family identification number of a font family you have already included.

To eliminate a reference to a font family, use GM_$FONT_FAMILY_EXCLUDE. There must be no references to the font family you want to exclude. Otherwise, the reference is not eliminated.

A font family identification may be referred to only if it has been included.

GM_$FONT_FAMILY_RENAME changes the font family file corresponding to this identification number.

6.5. A Program Including Text

The program in this section draws the design in Figure 6-1.

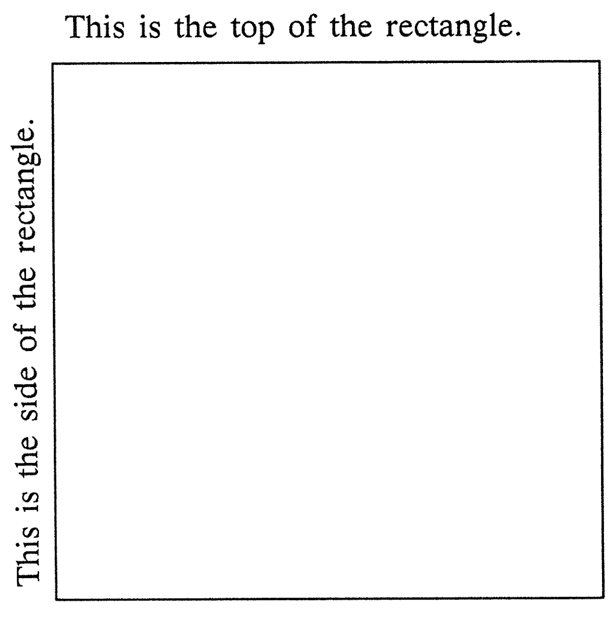

This is the top of the rectangle.
This is the side of the rectangle.

Figure 6-1. Inserting Text

This program draws an unfilled rectangle with the text strings "This is the top of the rectangle," and "This is the side of the rectangle." The routine GM_$FONT_FAMILY_INCLUDE specifies which font family to use in the metafile. You must create your own font families. For example, you can create a Display Manager file with the name font_families and place the names of the fonts you want to use in this file. In the file you could write the following font names that are pathnames to specific fonts:

```
  /sys/dm/fonts/f9x15
  /sys/dm/fonts/f7x13
  /sys/dm/fonts/f5x9
  /sys/dm/fonts/f5x7
```

The routine GM_$TEXT_SIZE specifies the maximum height of a character from the font family that you are using. The value of 14 specified in this program allows the use of text up to size 14. The default maximum text size is 10.

The routine GM_$TEXT_2D16 inserts a text string into the segment at the specified location. The first time that this routine is used, the second parameter (rotate) is listed as 0.0. This writes the text string horizontally ("This is the top of the rectangle"). The second time that this routine is called, the rotation is set at -90. This causes the text string to be written from bottom to top ("This is the side of the rectangle").

```
program draw_rectangle_text;

%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/gmr.ins.pas';
%include '/sys/ins/time.ins.pas';
%list;

CONST

    one_second = 250000;
    five_seconds = 5 * one_second;

VAR

    file_id             :  integer;
    segment_id          :  gm_$segment_id_t;
    st                  :  status_$t;
    pt1, pt2,point      :  gm_$point16_t;
    i                   :  integer32;
    bitmap_size         :  gm_$point16_t := [1024,1024];
    ffid                :  integer;
    pause               :  time_$clock_t;

BEGIN

    gm_$init                    { Initialize 2D GMR.}
         (gm_$direct
         ,1
         ,bitmap_size
         ,8
         ,st
         );
```

```
gm_$file_create             { Create and name a metafile. }
     ('gmfile'
     ,6
     ,gm_$overwrite
     ,gm_$1w
     ,file_id
     ,st
     );

gm_$segment_create           { Create and name a segment. }
     ('rectang_seg',
     sizeof('rectang_seg')
     ,segment_id
     ,st
     );

gm_$font_family_include       { Load the font family. }
     ( 'font_families'
     , SIZEOF('font_families')
     , gm_$pixel
     , ffid
     , st
     );

gm_$text_size
     ( 14.0
     , st
     );

point.x := 5;
point.y := 510;

gm_$text_2d16            { Display a line of text. }
     ( point, 0.0
     , 'This is the top of the rectangle.'
     , SIZEOF('This is the top of the rectangle.')
     , st
     );

point.x := 5;
point.y := 50;

gm_$text_2d16            { Display a line of text. }
     ( point
     , -90.0
     , 'This is the side of the rectangle.'
     , SIZEOF('This is the side of the rectangle.')
     , st
     );


                        { Define the coordinates of the }
pt1.x := 10;            { rectangle to be drawn. }
pt1.y := 30;
pt2.x := 400;
pt2.y := 500;

gm_$rectangle_16        { Draw the rectangle. }
     (pt1
        ,pt2
        ,false
        ,st
        );

   gm_$segment_close    { Close the segment. }
        (true
        ,st
        );

   gm_$display_file     { Display the file. }
        (st
        );

                       { Display the figure for five seconds. }
   pause.low32  := five_seconds;
   pause.high16 := 0;
   time_$wait
        ( time_$relative
        , pause
        , st
        );

   gm_$file_close       { Close the metafile. }
        ( true
        ,st
        );

   gm_$terminate        { Terminate 2D GMR. }
        ( st
        );

END.
```

6.6. Editing Fonts and Font Families

To change the names of font families already included in a metafile, you must open the metafile and use GM_$FONT_FAMILY_RENAME.

Three mechanisms are available for altering the form in which text appears on the screen:

- You can use a different font family in the metafile. To do this, open the metafile and use GM_$FONT_FAMILY_RENAME. This causes a different font family name to be associated with this metafile.

- You can change the list of fonts which make up a font family. To do this, open the ASCII file which lists the fonts in the font family. Edit this ASCII file using the text editor. This causes different fonts to be associated with this font family name.

- You can change the characters in the font. To create and edit stroke fonts, see the next section. To edit pixel fonts, use EDFONT, which allows you to interactively edit and view character font files. For a description of EDFONT, see the *DOMAIN System Command Reference* manual.

6.7. Creating Stroke Font Files

A stroke font file is a metafile. It is created and edited using normal metafile routines.

6.7.1. Defining Characters

The characters or icons of a stroke font are made up of primitive commands, like any other metafile segment. Each character has its own segment in a metafile. The set of characters (the font) is stored in a metafile.

You may define any or all of the 256 possible characters in a font, including the space character. The segment name for each character must be a one-character name. This name must be either the character that the segment defines or a nonprintable ASCII value for special icon definition.

A stroke font metafile may not contain attributes or instances, only primitives and tags. If any attribute or instance command is found in a stroke font segment, the specified character is treated as nonexistent.

When a text string is displayed, each character is displayed based on the following:

- Y = 0 is the baseline along which text is displayed.
- Y = 1000 is to be scaled to the current text size.
- The start of the next character is defined as the start of the old character, plus the horizontal size of the old character, plus an intercharacter spacing of 200.

6.7.2. Defining Character Width

The default width of a character is the maximum x value used in defining the segment (segment bounds). In a segment for an individual character, you can use a tag command to change the width of a character. A tag consists of WIDTH followed by a number to indicate the character width you want. (Separate the word WIDTH and the value by a space.) This tag command must precede any other commands in the segment. Descriptor tags in a stroke font file must be entered in the file in capital letters, for example, WIDTH.

The tag WIDTH is also used to define the space character (chr(32)), which contains only the WIDTH tag command. If a text string being displayed includes a nonexistent character (one for which there is no segment), the space character is displayed instead. If you have not defined a space character, nothing is displayed, and words within a string will run together.

6.7.3. Font Defaults

To set values applicable to every character and icon in the stroke font file, you can create an additional segment with the name DATA (the length of the segment name = 4). In this segment, only certain tag commands are recognized.

The following is a list of key words and their default values. You can change these values using tag commands in the segment DATA.

Key Word	Default Value
HEIGHT	1000
CHAR_OFFSET	200
MIN_SIZE	0

Definition of Terms

- HEIGHT: The maximum y value above the origin, in coordinates of the character segment. This value is used to scale text in this font to fit the current text size.
- CHAR_OFFSET: The x offset between the end of a character and the start of the next character, in coordinates of the character segment. This value is used for spacing between characters in a string.
- MIN_SIZE: The minimum height, in pixels, for which to use this font. If a transformation causes the character HEIGHT on the screen to be less than MIN_SIZE, this font is not used.

The DATA segment need not appear in any specific location in the stroke file. You may omit the DATA segment if you wish to use default values.

6.7.4. Limitations

The following limitations apply to the maximum number of font families and the maximum number of font files within families and overall.

You may specify a maximum of eight font families. Overall the font families, you may specify a maximum of 32 font files. If different font families specify the same font file, 2D GMR does not recognize the redundancy, so each reference counts as a separate font file.

A font family of stroke text may specify only two stroke text font files; a font family of pixel text may specify any number of pixel text font files (up to the overall limit of 32 font files).

6.8. A Procedure to Define a Font

This program example defines a stroke font. To define a font with equal spacing between characters, tag each character with an equal WIDTH and center each character definition between 0 and WIDTH in x. Adjust the CHAR_OFFSET accordingly.

To define a proportionally spaced font, set the minimum x value to 0 and assign a value to

WIDTH for each character. (Use uppercase for the value WIDTH). Alternatively, you can omit the specification of WIDTH. In this case, the maximum x of the character is the WIDTH, and the CHAR_OFFSET is used to separate the characters in a string.

The following examples show the use of tags with character and icon definition:

```
gm_$segment_create('A', 1, segment_id, st);
point_array[1].x := 0;
point_array[1].y := 0;
point_array[2].x := 400;
point_array[2].y := 1000;
point_array[3].x := 800;
point_array[3].y := 0;
gm_$polyline_2d16(3, point_array, false, false, st);
point_array[1].x := 200;
point_array[1].y := 500;
point_array[2].x := 600;
point_array[2].y := 500;
gm_$polyline_2d16(2, point_array, false, false, st);
gm_$segment_close(true, st);

gm_$segment_create(' ', 1, segment_id, st);
gm_$tag('WIDTH 800', 9, st);
gm_$segment_close(true, st);


gm_$segment_create('DATA', 4, segment_id, st);
gm_$tag('MIN_SIZE  15', 12, st);
gm_$segment_close(true, st);
```

6.9. Program With Stroke and Pixel Fonts

The following program loads a pixel font family file and a stroke font family file and then shifts back and forth between them using an attribute block and attribute class command. The purpose of this program is to illustrate the use of the two types of text. The attribute block and attribute class command provide an easy way to change text size. For a discussion of attribute blocks and attribute classes, see Chapter 13.

The first part of the program shows the effect of changing text size with pixel and stroke fonts One segment containing text is created and then redisplayed with different text sizes. The text size is changed using an attribute block.

The second part of the program creates a new segment and then instances the original text segment using GM_$INSTANCE_TRANSFORM_2D16. The instance command is replaced again and again with different angles of rotation to illustrate the effect on the text. The program shows that rotated pixel text snaps to the nearest 90 degrees, whereas stroke text rotates smoothly.

```
PROGRAM text;

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
```

```
%INCLUDE '/sys/ins/gmr.ins.pas';
%INCLUDE '/sys/ins/pfm.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';
%LIST;

CONST

    aclass1     = 1;
    second      = 500000;
    cos_delta   = COS( 0.25 );
    sin_delta   = SIN( 0.25 );

VAR

    status           : status_$t;
    sid_text         : gm_$segment_id_t;
    sid_top          : gm_$segment_id_t;
    file_id          : INTEGER;
    ffid_pixel       : INTEGER;
    ffid_stroke      : INTEGER;
    ablock_id        : INTEGER;
    p                : gm_$point16_t;
    q                : gm_$point16_t;
    dbounds          : gm_$boundsreal_t;
    i                : INTEGER;
    j                : INTEGER;
    text_size        : REAL;
    text_size_delta  : REAL;
    rotate           : gm_$rotate_real2x2_t;
    translate        : gm_$point16_t;

    pause            : time_$clock_t;

    PROCEDURE check
        ( IN    status  : status_$t
        );
    BEGIN
        IF status.all <> status_$ok
        THEN pfm_$error_trap( status );
        END;

BEGIN

    p.x := 1024;
    p.y := 1024;

    gm_$init                                    { Initialize the 2D GMR package. }
        ( gm_$direct
        , 1
        , p
        , 8
        , status
        );
    check( status );

    gm_$file_create                             { Create and name metafile. }
        ( 'gmfile'
        , 6
        , gm_$overwrite
    , gm_$1w
    , file_id
    , status
    );
check( status );

gm_$viewport_set_refresh_state          { Set viewport refresh state. }
    ( gm_$refresh_wait
    , status
    );
check( status );

gm_$font_family_include                 { Include pixel font family. }
    ( 'ff0'
    , 3
    , gm_$pixel
    , ffid_pixel
    , status
    );
check( status );

gm_$font_family_include                 { Include stroke font family. }
    ( 'ffs'
    , 3
    , gm_$stroke
    , ffid_stroke
    , status
    );
check( status );

gm_$ablock_create                       { Create an ablock. }
    ( 1
    , ablock_id
    , status
    );
check( status );

gm_$ablock_assign_display               { Ablock_id = aclass1. }
    ( aclass1
    , ablock_id
    , status
    );
check( status );

gm_$segment_create                      { Create a text segment. }
    ( ''
    , 0
    , sid_text
    , status
    );
check( status );

gm_$aclass                              { Add an aclass command }
    ( aclass1
    , status
    );
check( status );

p.x := - 5;
```

```
p.y := - 5;
q.x := + 5;
q.y := + 5;

gm_$rectangle_16                        { Add an unfilled rectangle. }
    ( p
    , q
    , FALSE
    , status
    );
check( status );

p.x := + 10;
p.y :=    0;

gm_$text_2d16                           { Add Left to Right text }
    ( p
    , 0.0
    , 'Left to Right'
    , 13
    , status
    );
check( status );

p.x :=    0;
p.y := - 10;

gm_$text_2d16                           { Add Top to Bottom text. }
    ( p
    , 90.0
    , 'Top to Bottom'
    , 13
    , status
    );
check( status );

p.x := - 10;
p.y :=    0;

gm_$text_2d16                           { Add Right to Left text. }
    ( p
    , 180.0
    , 'Right to Left'
    , 13
    , status
    );
check( status );

p.x :=    0;
p.y := + 10;

gm_$text_2d16                           { Add Bottom to Top text. }
    ( p
    , -90.0
    , 'Bottom to Top'
    , 13
    , status
    );
check( status );
```

```
gm_$segment_close                      { Close the segment. }
    ( true
    , status
    );
check( status );

dbounds.xmin := - 50.0;
dbounds.ymin := - 50.0;
dbounds.xmax := + 50.0;
dbounds.ymax := + 50.0;

pause.low32  := second DIV 4;
pause.high16 := 0;

text_size := 10;
text_size_delta := 1.0;

{ * * Illustrate different text sizes with pixel and stroke text. * * }

FOR j := 1 TO 2
DO BEGIN

    IF j = 1
    THEN gm_$ablock_set_font_family    { Set ablock to pixel font family. }
        ( ablock_id
        , ffid_pixel
        , status
        )
    ELSE gm_$ablock_set_font_family    { Set ablock to stroke font family. }
        ( ablock_id
        , ffid_stroke
        , status
        );
    check( status );

    FOR i := 1 TO 20
    DO BEGIN

        IF      text_size >= 10.0
        THEN text_size_delta := - ABS( text_size_delta )
        ELSE IF text_size <= ABS( text_size_delta )
        THEN text_size_delta := + ABS( text_size_delta );

        text_size := text_size + text_size_delta;

        gm_$ablock_set_text_size        { Change ablock text size. }
            ( ablock_id
            , text_size
            , status
            );
        check( status );

        gm_$display_file_PART           { Display the file. }
            ( dbounds
            , status
            );
        check( status );

        time_$wait                      { Admire it for a momemt. }
```

```
            ( time_$relative
            , pause
            , status
            );
        check( status );

        END;

    END;

gm_$segment_create                       { Create top segment. }
    ( ''
    , 0
    , sid_top
    , status
    );
check( status );

rotate.xx := 1.0;                        { Define identity matrix. }
rotate.xy := 0.0;
rotate.yx := 0.0;
rotate.yy := 1.0;

translate.x := 0;                        { Zero translation }
translate.y := 0;

gm_$instance_transform_2d16              { Instance text segment into }
    ( sid_text                           { top segment. }
    , rotate
    , translate
    , status
    );
check( status );

gm_$modelcmd_set_mode                    { Go into replace mode. }
    ( gm_$modelcmd_replace
    , status
    );
check( status );

{ * * Illustrate different text angles with pixel and stroke text. * * }

FOR j := 1 TO 2
DO BEGIN

    IF j = 1
    THEN gm_$ablock_set_font_family      { Set ablock to pixel font family. }
        ( ablock_id
        , ffid_pixel
        , status
        )
    ELSE gm_$ablock_set_font_family      { Set ablock to stroke font }
        ( ablock_id                       { family. }
        , ffid_stroke
        , status
        );
    check( status );

    FOR i := 1 TO 40
    DO BEGIN

        WITH rotate                     { Increment the rotation matrix. }
        DO BEGIN
            xx := cos_delta * xx + sin_delta * xy;
            yx := cos_delta * yx + sin_delta * yy;
            xy := - yx;
            yy := + xx;
            END;

        gm_$instance_transform_2d16    { Change the angle of the instance }
            ( sid_text                 {  transform }
            , rotate
            , translate
            , status
            );
        check( status );

        gm_$display_file_PART           { Display the file. }
            ( dbounds
            , status
            );
        check( status );

        time_$wait                      { Admire it for a moment. }
            ( time_$relative
            , pause
            , status
            );
        check( status );

        END;

    END;

gm_$segment_close                       { Close the top segment. }
    ( TRUE
    , status
    );
check( status );

gm_$file_close                          { Close the file. }
    ( true
    , status
    );
check( status );

gm_$terminate                           { Terminate the 2D GMR package. }
    ( status
    );
check( status );

END.
```

Chapter 7
Using Segment Characteristics

This chapter describes the use of the primary segment in displaying a metafile. Characteristics that can be associated with a segment are discussed and illustrated with a program. The formats for coordinate data are described.

7.1. Primary Segment

Functions:

```
GM_$FILE_SET_PRIMARY_SEGMENT
GM_$FILE_INQ_PRIMARY_SEGMENT
```

The segments in the metafile have a hierarchical structure. The primary segment can be thought of as the root for the hierarchy of segments in the metafile. As such, the primary segment is assumed to be the start of the picture. When the routine GM_$DISPLAY_FILE is called, the primary segment is displayed.

The first segment you create becomes the primary segment. In Figure 7-1, the primary segment is "scene." When you display "scene," you see the entire picture. The segments are instanced according to the hierarchy established by the primary segment.

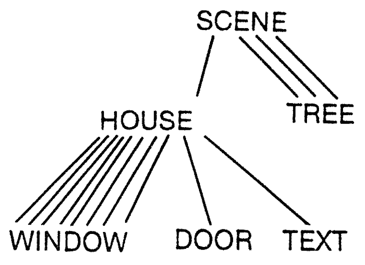


Figure 7-1. Hierarchical Structure and the Primary Segment

Using GM_$FILE_SET_PRIMARY_SEGMENT, you can specify that you want another segment as the primary segment. For example, with "house" as the primary segment in Figure 7-1, you see the following upon display: house, door, eight windows, and sign. You do not see any trees.

If you instance the primary segment from a segment that is not itself instanced, the primary segment is changed to the instancing segment. If you instance the primary segment from a segment that is itself instanced, the root of the instancing tree becomes the primary segment (see Figure 7-1).

Use GM_$FILE_SET_PRIMARY_SEGMENT to change the primary segment number. Use GM_$FILE_INQ_PRIMARY_SEGMENT to retrieve the number of the primary segment.

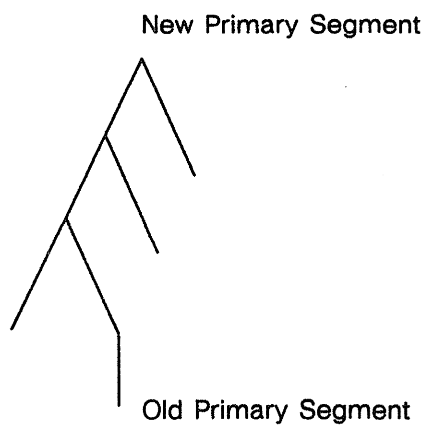


Figure 7-2. Instancing and the Primary Segment

7.2. Setting Segment Characteristics

Functions:

```
GM_$SEGMENT_SET_VISIBLE
GM_$SEGMENT_INQ_VISIBLE
GM_$SEGMENT_SET_PICKABLE
GM_$SEGMENT_INQ_PICKABLE
GM_$SEGMENT_SET_TEMPORARY
GM_$SEGMENT_INQ_TEMPORARY
```

Segment characteristics, such as visible and pickable, are associated with a segment rather than being controlled by commands contained within a segment. The visible value lets you specify that a segment be visible or invisible when you display the file. The pickable value lets you specify that a segment be eligible or ineligible for selection during a search. The name of the segment, which you assign when you call GM_$SEGMENT_CREATE, is also a segment characteristic.

GM_$SEGMENT_SET_VISIBLE assigns a value at which the specified segment can be seen when you display the file. This value is used with the visible threshold and visible mask during display operations. This enables you to display a picture without segments of it that may clutter it. For example, you may want to see a picture with or without text. You can place text in a separate segment and then change the visible value of that segment to make it visible or invisible.

GM_$SEGMENT_INQ_VISIBLE returns the visible value of the specified segment.

GM_$SEGMENT_SET_PICKABLE assigns a pickable value to the specified segment. This pickable value is used with the pick threshold and pick mask during pick operations (see Chapter 10). GM_$SEGMENT_INQ_PICKABLE returns the pickable value of the specified segment.

GM_$SEGMENT_SET_TEMPORARY sets the current segment as temporary or permanent. A temporary segment is deleted when the file is closed. A temporary segment is useful for picture data that you want to display but not store. This allows you to add a graphic element, such as enclosing boxes or a superimposed grid, which you do not want to store in the metafile.

GM_$SEGMENT_INQ_TEMPORARY indicates whether the current segment is set to temporary or permanent.

7.3. Coordinate Data Types

```
Functions:

GM_$DATA_COERCE_SET_REAL
GM_$DATA_COERCE_INQ_REAL
```

For your convenience, the 2D GMR package has multiple formats for your input of coordinate data. For efficiency, the 2D GMR package converts, or coerces, the coordinate data you supply to a different format. You need to know about 2D GMR data storage conventions because inappropriate mixing of formats can result in a loss of precision in stored data. However, you can use coordinate data types in the following ways without concern for loss of precision.

You will not lose precision if:

- You only supply 16-bit integer data to a segment.
- You only supply 32-bit integer data to a segment, and store it in 32-bit segment-exponent format.

If you wish to supply data in other forms or combinations, you should be aware of the conventions described on the paragraphs below.

You may supply coordinate data to the modeling routines of the 2D GMR package as 16- or 32-bit integers, or as single-precision real numbers. Different routines exist to accept data in these different forms. These routines are generally referred to in this document as groups. For example, GM_$POLYLINE_2D[16,32,REAL] refers to a group of three routines:

```
GM_$POLYLINE_2D16
GM_$POLYLINE_2D32
GM_$POLYLINE_2DREAL
```

Each of these routines differs only in the data types of its coordinate parameters. Each routine with a data type (16, 32, or REAL) in its name indicates the type of variable or array you use to supply coordinate data to the 2D GMR package.

There are two internal data storage formats: 16-bit segment-exponent and 32-bit segment-exponent. In either format, the following occurs:

- All data within a segment is stored with a common binary exponent. This is stored as a characteristic of the segment.
- 16-bit or 32-bit signed integers are used to store values to be multiplied by the common binary exponent. The choice is made by the data type specified in the name of the routine, for example, GM_$POLYLINE_2D16 or GM_$POLYLINE_2D32.

You can store 16-bit and 32-bit data in the same segment. The 16 most significant bits of 32-bit data correspond to the 16 bits of 16-bit data, and the 16 least significant bits simply add more precision. This is analogous to the extra bits of precision provided by using double-precision real numbers instead of single-precision real numbers.

When you insert numbers larger than those you originally supplied, the existing values may be shifted. When you supply new values that are too large to be stored using the current exponent, the 2D GMR package changes the exponent, and shifts all existing values to conform to this new exponent.

You can use GM_$DATA_COERCE_SET_REAL to indicate that coordinate data supplied in one form is to be coerced into another form for storage. For example, you can send data to the package as real variables, but store data in the file in 32-bit segment-exponent format. To retrieve the storage format to which real coordinates are being converted, use GM_$DATA_COERCE_INQ_REAL.

Currently, you must use GM_$DATA_COERCE_SET_REAL(GM_$32,STATUS) prior to calling any GM_$...REAL command. The package does not currently store real data as such.

One feature of this storage format is that you may mix integer and real data within a segment. For example, you can insert a coordinate value (5.5) into a segment that previously contained integers, without adding to the processing required to display the integer data.

You should avoid storing data in the metafile with more precision than you need because the more complex calculations affect performance. You may mix data storage types within a segment, but not within individual commands. For example, you may find it necessary to use 32-bit storage format for coordinate transformations to preserve precision through multiple coordinate transformations; but 16-bit storage format may be adequate for the primitive commands within these instanced segments.

You can lose precision of original data in these cases:

- When you coerce real data to 16-bit segment-exponent format for storage, the least significant bits are dropped immediately.

- When you have stored 16-bit integer coordinate data in 16-bit segment-exponent format, and subsequently insert 32-bit integer or real data of absolute value larger than 32767. Low order bits of the original data will then be truncated. For example, 0 and 1 both become 0.

- When you have stored 16-bit or 32-bit integer coordinate data in 32-bit segment-exponent format, and subsequently insert real data of absolute value greater than the largest 32-bit signed integer.

You will not lose precision if:

- You only supply real data to a segment, coerce it to 32-bit segment-exponent format for storage, and supply data which ranges only over a factor of 128.

Chapter 8
The Displaying Process

This chapter describes the display environment and coordinate systems used with the graphics metafile package. Viewing routines are presented with a sample program to illustrate their use.

8.1. Hardware and Coordinate Systems

```
Functions:

GM_$INQ_CONFIG
GM_$INQ_BITMAP_SIZE
GM_$COORD_SEG_TO_BITMAP_2D
GM_$COORD_BITMAP_TO_SEG_2D
GM_$COORD_PIXEL_TO_SEG_2D
GM_$COORD_SEG_TO_PIXEL_2D
```

The 2D GMR package is generally independent of the display environment. This means that you can run most programs that include 2D GMR routines on any DOMAIN node without modifying the program. The 2D GMR package rescales viewports and views according to the size of the GM bitmap, so that programs containing viewing routines will display all viewports for any GM bitmap.

When you use the 2D GMR routines, you can easily change program execution from one display mode to another by changing one option in the initialization routine GM_$INIT.

You can determine the configuration of the display device using GM_$INQ_CONFIG. You can use this information to assign different attributes to a color display than to a monochromatic display.

GM_$INQ_CONFIG returns the current configuration of the display device. Possible values are the following:

GM_$BW_800x1024	4-bit two-board black-and-white portrait.
GM_$BW_1024x800	4-bit two-board black-and-white landscape.
GM_$COLOR_1024x1024x4	4-bit two-board color configuration.
GM_$COLOR_1024x1024x8	8-bit three-board color configuration.
GM_$COLOR_1024x800x4	4-bit two-board color configuration.
GM_$COLOR_1024x800x8	8-bit two-board color configuration.

GM_$INQ_BITMAP_SIZE returns the size of the GM bitmap in pixels, and the number of planes in the GM bitmap.

GM_$COORD_BITMAP_TO_SEG_2D converts bitmap coordinates to segment coordinates of the viewport primary segment in the current viewport.

GM_$COORD_SEG_TO_BITMAP_2D converts segment coordinates (of the viewport primary segment in the current viewport) to bitmap coordinates.

GM_$COORD_PIXEL_TO_SEG_2D converts GPR bitmap coordinates used in within-GPR mode to segment coordinates, using a specified transformation.

GM_$COORD_SEG_TO_PIXEL_2D converts within-GPR segment coordinates to GPR bitmap coordinates, using a specified transformation.

8.2. Display Coordinates and Mode

This section gives an overview of display coordinates, multiple viewports, and view changes. Subsequent sections in this chapter describe the routines for these functions in more detail.

When you use the 2D GMR package in borrow, direct, and main-bitmap display modes, you do not have to refer to pixel coordinates in a bitmap. The 2D GMR package handles all the transformations from world coordinates in the segment to pixels on the screen. The only exception to this is that you specify the size of the GM bitmap in the routine GM_$INIT. After that specification, the only references to coordinates on the display are in terms of fractions of the size of the GM bitmap.

The term fraction-of-bitmap refers to the bitmap that was established as a result of the initialization command. The coordinates are from 0 to 1 left to right and from 0 to 1 bottom to top. X increases to the right; y increases up. Coordinate locations within the GM bitmap are expressed as fractions of the GM bitmap's size, whether it is square or nonsquare.

Every time you display a segment in the viewport, 2D GMR calculates the transformation of the world coordinates to bitmap pixels for each coordinate and processes the commands in the metafile. The 2D GMR package also adjusts the transformations for any changes that result from the process.

In addition, if you change a view by using viewing commands, by popping windows, or by changing the size of the DM window, the 2D GMR package recalculates all transformation variables in a way that is transparent to the user. All commands that adjust the size of viewports or that deal with locator input use window-independent coordinates. These are world or fractional coordinates, not device or pixel coordinates.

You may subdivide the GM bitmap into multiple viewports, but overlapping viewports are not supported. As a viewport includes its border, the borders may not overlap either. When the 2D GMR package is initialized, one viewport is created, and it is assigned the number one. This viewport is defined to fill the GM bitmap; that is, it is given bounds of (0,0) through (1,1), in terms of fraction-of-bitmap coordinates. This viewport becomes the current viewport. Before creating any additional viewports, you must change the bounds of viewport 1 to make room on the screen for another viewport.

To change the dimensions of the viewport, use GM_$VIEWPORT_SET_BOUNDS. To redefine the viewport with this routine, you must provide the coordinates of any two diagonally opposite corners. Note that if the GM bitmap is not square, the units in the x and y directions are different. For example, if the bitmap is 1000(x) by 500(y), 0.01 means 10 pixels in the x direction, but only 5 pixels in the y direction.

An image displayed in the viewport is not physically moved on the screen as a result of a GM_$VIEWPORT_SET_BOUNDS command. That is, the transformation from world coordinates to bitmap coordinates does not change. If you make the viewport smaller, the object is displayed the same size on the screen. However, you will see less of it or less blank space around it.

To change the size of what is displayed on the screen, you can use one of several techniques. One way is to use GM_$VIEWPORT_SET_BOUNDS, followed by GM_$DISPLAY_SEGMENT. The segment will be displayed into that viewport and fill the new bounds.

Once you have used GM_$VIEWPORT_SET_BOUNDS to reduce viewport 1, you can call GM_$VIEWPORT_CREATE. The current viewport is the last viewport that you created or selected. It is the implied viewport for GM_$DISPLAY_FILE, GM_$DISPLAY_SEGMENT, GM_$VIEWPORT_SET_BOUNDS, and GM_$VIEWPORT_REFRESH. Before you go from one viewport to another, get the number of the current viewport so that you can return to it easily. GM_$VIEWPORT_INQ_CURRENT returns the number of the current viewport.

You may want to write an application program that does not require tracking the units on the display. Instead, you have the program display the whole segment and allow the user to specify scaling parameters by using function keys or a locator input device to move the data on the screen. To do this, you use the routines GM_$VIEW_TRANSLATE, GM_$VIEW_SCALE, and GM_$VIEW_TRANSFORM.

GM_$VIEW_TRANSLATE does not change the scale of the displayed data; the routine moves the image on the screen, translating the view in the current viewport by (x,y). X and y are expressed in the view transform as fractions of the GM bitmap, not fractions of the viewport.

If the viewport fills the entire screen, you need not be concerned about the difference between fractions of the GM bitmap and the viewport. If you have a smaller viewport, you will have to get the size of the viewport. For example, in an application program, you may want to define the box-left button to slide the image to the left by half the viewport. You must first call GM_$VIEWPORT_INQUIRE_BOUNDS to get the bounds of the current segment. You can then calculate the difference between xmax and xmin for that viewport, divide by 2, and use that as input to the GM_$VIEW_TRANSLATE routine.

GM_$VIEW_SCALE and GM_$VIEW_TRANSFORM change the scale of what is viewed in the current viewport. These routines can either scale or transform the view in the current viewport. In both of these routines, the specified point in the GM bitmap is fixed during the operation. When the user moves the cursor and indicates increasing or diminishing the size of the image, the locator input data is returned in fraction-of-bitmap coordinates. This means that all points are scaled around this fixed point to allow either a scale factor or an arbitrary 2D transformation around this fixed point.

With GM_$VIEW_TRANSFORM, you use a two-dimensional transformation (2 by 2 transformation in an array of four real values).

You can avoid using the transformation by calling GM_$VIEW_SCALE. The following example shows rescaling the screen by a factor "scale" and moving the point (point.x,point.y) to the center of the viewport, all in one operation. (The coordinates on the screen are expressed as fractions of the GM bitmap.) This example is for a 50-degree rotation:

```
{ Assumes scale not equal to 1.0 }

GM_$VIEWPORT_INQ_BOUNDS (vbounds, status);
vcenter_x := 0.5 * (vbounds.xmax + vbounds.xmin);
vcenter_y := 0.5 * (vbounds.ymax + vbounds.ymin);
point1.x := (vcenter_x - point.x * scale)/(1.0 - scale);
point1.y := (vcenter_y - point.y * scale)/(1.0 - scale);
GM_$VIEW_SCALE(scale, point1,status);
```

These routines allow relative view changes without tracking the view transformation or the displayed image. For example, the user can move the cursor to a viewport; the locator of input data tells the 2D GMR package what the coordinates are, in fractions of bitmap coordinates, and what the viewport number is. With the viewport number from the input data, the 2D GMR package can make that viewport the current viewport and then use the fixed point as a reference for scaling or transformation.

8.3. Using Multiple Viewports

Functions:

GM_$VIEWPORT_CLEAR
GM_$VIEWPORT_CREATE
GM_$VIEWPORT_SET_BOUNDS
GM_$VIEWPORT_INQ_BOUNDS
GM_$VIEWPORT_SELECT
GM_$VIEWPORT_DELETE
GM_$VIEWPORT_INQ_CURRENT
GM_$VIEWPORT_MOVE
GM_$VIEWPORT_SET_BORDER_SIZE
GM_$VIEWPORT_INQ_BORDER_SIZE

When you initialize the 2D GMR package in direct, borrow, and main-bitmap modes, the package does the following:

- Creates one viewport.
- Makes the viewport fill the GM bitmap.
- Assigns number 1 to the viewport.
- Makes number 1 the current viewport.

To use multiple viewports, you create additional viewports with GM_$VIEWPORT_CREATE. The 2D GMR package assigns numbers to viewports as they are created.

The current viewport is the last viewport created or selected. The current viewport is changed each time you call GM_$VIEWPORT_CREATE or GM_$VIEWPORT_SELECT. GM_$VIEWPORT_CLEAR clears the current viewport. Only planes enabled by the current value of the plane mask are affected.

Several routines which interact with viewports operate on the current viewport. These routines include GM_$VIEWPORT_SET_BOUNDS, GM_$VIEWPORT_REFRESH, GM_$DISPLAY_FILE, and GM_$DISPLAY_SEGMENT.

GM_$VIEWPORT_INQ_BOUNDS returns the bounds of the current viewport as fractions of the GM bitmap. Space outside of all viewports is empty.

GM_$VIEWPORT_MOVE translates the current viewport, carrying the view with it. The translation is expressed in fractions of the display bitmap size.

GM_$VIEWPORT_INQ_CURRENT returns the number of the current viewport. When GM_$INPUT_EVENT_WAIT collects locator data, it also returns the number of the viewport to tell what viewport you are in.

GM_$VIEWPORT_DELETE deletes a viewport. Because viewports may not overlap, you must delete all but one viewport if a single viewport is to fill the entire GM bitmap. If you delete the current viewport, there is no current viewport and you must select or create a current viewport before calling routines that operate on the current viewport.

GM_$VIEWPORT_SET_BORDER_SIZE sets the border size of the current viewport to the specified values, either in pixels or in fraction-of-bitmap coordinates. This routine sets sizes of the four edges independently, for each viewport.

GM_$VIEWPORT_INQ_BORDER_SIZE returns the border size of the current viewport, either in pixels or in fraction-of-bitmap coordinates.

The default border type is in pixels, and the default width is 1,1,1,1. Viewport borders are drawn with color value 1 for compatibility with monochrome nodes. Also for this compatibility, the 2D GMR package sets the color map for color value 1 to white.

With a color node, you may want to use the viewport background color, instead of a border, to differentiate viewports from the overall display or the window background. Changing the color map to black is usually not practical because the cursor is also set to color value 1. An alternative is to create the viewport, set the border width to 0 pixels, and then refresh the viewport.

8.4. Segment Visibility Criteria

Functions:

```
GM_$VISIBLE_SET_MASK
GM_$VISIBLE_INQ_MASK
GM_$VISIBLE_SET_THRESHOLD
GM_$VISIBLE_INQ_THRESHOLD
```

You can establish two criteria for segment visibility. You may use a threshold to eliminate segments with visible values too small. You may also use a mask to eliminate segments missing certain bits in their visible values.

To assign visible values to segments, use GM_$SEGMENT_SET_VISIBLE (see Section 7.2).

A segment is displayed only if it meets both the visible mask and threshold criteria. The visible mask criterion requires that at least one bit be "1" in both the segment's visible value and the visible mask. The visible threshold criterion requires that the segment's visible value be greater than the visible threshold.

One use for this pair of methods is to set the lowest bit (1) in the visible value in all segments containing, for example, a floor plan; the next lowest bit (2) containing data for ducting; and and the next bit (4) containing data for electrical systems. Higher bits can be used to give larger visible values to larger segments.

With the above settings, visible masks with the following values perform in this way:

```
1 displays only the floor plan
2 displays only ducting
5 displays the floor plan and electrical data
7 displays all three groups of data
```

If a segment does not satisfy both of the segment visibility criteria, none of that segment is displayed. Any segment which it instances is not checked for visibility and is not displayed.

In borrow, direct, and main-bitmap modes, you may assign separate visible mask and threshold values to each viewport. Thus, different viewports can display separate parts of a data base. Within-GPR mode has only one visible mask and one visible threshold value.

GM_$VISIBLE_SET_THRESHOLD establishes a minimum segment visible value. Segments with smaller visible values are not displayed. The default value is 1, in which case all segments of nonzero visible value satisfy the threshold criterion.

GM_$VISIBLE_SET_MASK allows you to base segment visibility on individual bits within the visible value for each segment. If any bit is 1 in both the mask and the segment visible number, the segment may be visible. The default value is 16#7FFFFFFF, in which case all segments of nonzero visible value satisfy the mask criteria.

GM_$VISIBLE_INQ_THRESHOLD and GM_$VISIBLE_INQ_MASK return the current value of these parameters for the specified viewport.

8.5. Displaying a File/Segment

Functions:

```
GM_$DISPLAY_FILE
GM_$DISPLAY_SEGMENT
GM_$DISPLAY_SEGMENT_GPR_2D
```

You can display the entire file or segment. The picture is automatically centered in the current viewport, with a scale calculated so that 95% of the viewport is filled in one dimension amd does not overflow the viewport in the other.

A segment from one file can be displayed in one viewport. A segment from another open file can be displayed in another viewport. But only one segment may be displayed in any viewport, and it may not instance segments in other files. To display the contents of more than one segment in the single viewport, build a new segment including instances of the other segments.

GM_$DISPLAY_FILE displays the entire current file in the current viewport. The primary segment of the file is displayed.

GM_$DISPLAY_SEGMENT displays the specified segment, but not the entire file, in the current viewport.

GM_$DISPLAY_SEGMENT_GPR_2D is used in within-GPR mode only. It displays the specified segment, with the specified transformation, in the current GPR-specified bitmap (see Chapter 11).

8.6. Displaying Part of a File/Segment

Functions:

```
GM_$DISPLAY_FILE_PART
GM_$DISPLAY_SEGMENT_PART
```

Only one segment may be displayed per viewport, but you can specify what part of that segment is displayed or change the part of that segment that is displayed. You may want to see only part of a graphic image you have developed. For example, you may want to see only the wheel of a car, not the entire body of the car that you have been modeling.

To get part of an image in a view, you use GM_$DISPLAY_SEGMENT_PART or GM_$DISPLAY_FILE_PART to specify in segment coordinates the part of the segment (or file) you want displayed. That part of the segment (or file) is centered in the current viewport with a scale automatically set so that the specified part of the file is displayed as follows: One of the two dimensions fills the viewport, and the other dimension does not overflow the viewport.

This allows you to look at the entire file or segment in one viewport and a smaller part of a file or segment in another viewport. The same file or segment can appear in different viewports (see Figure 4-1.

You may want the contents of a segment to be much smaller than the viewport. This is possible because the 2D GMR package uses the rectangular bounds that you specify, not the current bounds of the coordinate data within the segment. In this way, you can override this coordinate data based on coordinates within the segment.

The 2D GMR package then sets the view transformation from world coordinates of that segment to display pixels so that one dimension fills the current viewport 100% (not 95%). The other dimension is centered in the viewport and does not overflow it. This means that if the aspect ratio of the region you defined is different than the aspect ratio of the viewport, you will see more than the area you requested. It will continue to display all the way out to the edge of the viewport. But you are guaranteed that this entire rectangular bounded region will be viewed in the viewport, and one dimension or the other will fill the entire viewport.

GM_$DISPLAY_FILE_PART displays part of the current file in the current viewport. Bounds are in segment coordinates of the primary segment.

GM_$DISPLAY_SEGMENT_PART displays part of the specified segment in the current viewport.

8.7. Changing the View

Functions:

```
GM_$VIEW_TRANSLATE
GM_$VIEW_SCALE
GM_$VIEW_TRANSFORM
GM_$VIEW_TRANSFORM_RESET
```

Changing the view causes the picture in the current viewport to be redisplayed, with the picture translated or scaled.

GM_$VIEW_TRANSLATE translates the view in the current viewport by (x,y) in bitmap coordinates. The amount of translation is expressed in fractions of the size of the GM bitmap.

GM_$VIEW_SCALE rescales the display in the current viewport, multiplying the current view transformation by the specified scale factor. The point (x,y) on the screen is kept fixed during this rescaling. This point is expressed in bitmap coordinates, that is, fractions of the size of the GM bitmap.

GM_$VIEW_TRANSFORM rescales the display in the current viewport, multiplying the current view transformation by the specified transform factor. The point (x,y) on the screen is kept fixed during this rescaling. This point is expressed in bitmap coordinates, that is, fractions of the size of the GM bitmap.

GM_$VIEW_TRANSFORM_RESET restores the view transformation in the current viewport to the value that was assigned to it at the time of the last call to GM_$DISPLAY_FILE[_PART] or GM_$DISPLAY_SEGMENT[_PART], adjusted for any GM_$VIEWPORT_MOVE calls or any changes to the GM bitmap. This allows you to manipulate the view experimentally and still restore it to its form at the time you displayed it.

8.8. Refreshing the Display

Functions:

```
GM_$DISPLAY_REFRESH
GM_$VIEWPORT_REFRESH
GM_$REFRESH_SET_ENTRY
```

These routines enable an application program to keep up with changes to a file or segment. The routines are especially useful after you have made multiple changes to a file or segment.

GM_$DISPLAY_REFRESH updates the display in all viewports, except viewports in the GM_$REFRESH_INHIBIT refresh state (see Section 10.2).

GM_$VIEWPORT_REFRESH updates the display in the current viewport. This routine redisplays the contents of the viewport and is useful after you have made multiple changes to a file or segment.

GM_$REFRESH_SET_ENTRY specifies a user-defined routine to be called when the display is refreshed as a result of using a DM refresh window command or pressing <POP>.

8.9. Program to Change the View

This program opens and displays an existing file and then changes the view scale to shrink the viewport. (For an existing file, you may use the program in Appendix D). As viewports may not overlap, this makes room for a second viewport. The second viewport is created, and the file is displayed in it. The view scale is then changed for the second viewport.

To illustrate the ease of switching between viewports, the first viewport it selected as the current viewport. This viewport is moved and made smaller to allow for the creation of a third viewport. The file is displayed in this third viewport, and the view scale is changed.

```
PROGRAM course2;

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/gmr.ins.pas';
%LIST;

VAR
    status       : status_$t;

    name         : name_$pname_t;
    length       : INTEGER;

    vpid2        : INTEGER;
    vpid3        : INTEGER;
    file_id      : INTEGER;

    rtransl      : gm_$pointreal_t;
    b            : gm_$boundsreal_t;

    bitmap_size  : gm_$point16_t;

    i            : INTEGER;

BEGIN

    WRITE( 'File name:  ' );    { Input the name of the file to be displayed, }
    READLN( name );               for example, PROGRAM hotel. }

    length := LASTOF( name );
    WHILE ( name[ length ] = ' ' ) AND ( length > 0 )
    DO length := length - 1;

    bitmap_size.x := 1024;
    bitmap_size.y := 1024;
    gm_$init
        ( gm_$direct
        , stream_$stdout
        , bitmap_size
        , 8
        , status
        );

    gm_$file_open
        ( name
        , length
        , gm_$r
        , gm_$1w
        , file_id
        , status
        );

    gm_$display_file                    { Now display the file. }
        ( status
        );

    rtransl.x := 0.0;                   { Change the view scale. }
    rtransl.y := 1.0;
    gm_$view_scale
        ( 0.25
        , rtransl
        , status
        );

    FOR i := 1 TO 10
    DO gm_$view_scale
        ( 1.05
        , rtransl
        , status
        );

    rtransl.x := 0.1;
    rtransl.y := 0.8;
    FOR i := 1 TO 10
    DO gm_$view_scale
        ( 0.92
        , rtransl
        , status
        );

    b.xmin := 0.0;                      { Shrink the viewport. }
    b.ymin := 0.6;
    b.xmax := 0.38;
    b.ymax := 1.0;
    gm_$viewport_set_bounds
        ( b
        , status
        );

    b.xmin := 0.4;                      { Create a second viewport. }
    b.ymin := 0.0;
    b.xmax := 1.0;
    b.ymax := 0.6;
    gm_$viewport_create
        ( b
        , vpid2
        , status
        );

    gm_$display_file             { Display the file in the second viewport. }
        ( status
        );

    rtransl.x := 0.7;        { Change the view scale in the second viewport. }
    rtransl.y := 0.3;
```

```
FOR i := 1 TO 10
DO gm_$view_scale
    ( 1.05
    , rtransl
    , status
    );

FOR i := 1 TO 3
DO gm_$view_scale
    ( 0.85
    , rtransl
    , status
    );

gm_$viewport_select                { Switch back to the first viewport. }
    ( 1
    , status
    );

rtransl.x := 0.0;
rtransl.y := 0.9;
FOR i := 1 TO 10
DO gm_$view_scale
    ( 1.05
    , rtransl
    , status
    );

FOR i := 1 TO 3
DO gm_$view_scale
    ( 0.9
    , rtransl
    , status
    );

gm_$viewport_select          { Switch back to the second viewport. }
    ( vpid2
    , status
    );

rtransl.x := 0.0;                 { Translate the second viewport. }
rtransl.y := -0.4;
gm_$view_translate
    ( rtransl
    , status
    );

b.xmin := 0.4;                     { Shrink the second viewport. }
b.ymin := 0.4;
b.xmax := 1.0;
b.ymax := 1.0;
gm_$viewport_set_bounds
    ( b
    , status
    );

b.xmin := 0.0;                     { Create a third viewport. }
b.ymin := 0.0;
b.xmax := 1.0;
```

```
b.ymax := 0.2;
gm_$viewport_create
    ( b
    , vpid3
    , status
    );

gm_$display_file        { Display the file in the third viewport. }
    ( status
    );

gm_$viewport_inq_bounds
    ( b
    , status
    );

rtransl.x := 0.9;      { Change the view scale in the third viewport. }
rtransl.y := 0.1;
FOR i := 1 TO 20
DO gm_$view_scale
    ( 0.9
    , rtransl
    , status
    );

gm_$file_close                     { Close the file. }
    ( true
    , status
    );

gm_$terminate
    ( status
    );

END.
```

Chapter 9
Developing Interactive Applications

This chapter provides an overview of functions used for interactive editing. The routine for changing the editing mode is described and illustrated with a sample program. Other interactive functions are described in more detail in Chapter 10.

9.1. Making Your Application Easy to Use

To make your application program easy to use, the 2D GMR package provides the tools for interaction between your program and your user. The 2D GMR package has routines to establish editing modes, to accept input, and to refresh the display to accommodate change to an image and the screen.

The routines GM_$VIEWPORT_SET_REFRESH_STATE and GM_$VIEWPORT_INQ_REFRESH_STATE allow you to control the frequency at which the display in a viewport is refreshed. With these routines, you can change the metafile and have the package automatically update one or more viewports to incorporate these changes, without calling a refresh routine. One use of this feature is in a rubberbanding procedure when you are trying to find the right place to put a line (see Sections 9.2 and 10.2).

The 2D GMR package has routines to control cursor activity, position, and appearance. These routines help establish an easy user interface. The cursor routines are described in Section 10.3).

The routines GM_$INPUT_ENABLE and GM_$INPUT_DISABLE enable graphics programs to accept input from various input devices. The input routines can be used to synchronize program execution around input events. These input routines function only in direct mode and in borrow mode. In within-GPR mode, you must use GPR input routines (see Section 11.1.2).

When you use input routines, you may specify whether the process is to wait until an enabled event occurs or to return a GM_$NO_EVENT event type if no event has occurred. To do this, you use the routine GM_$INPUT_EVENT_WAIT. For a description of these routines, see Sections 10.4.1 and 10.4.2.

Pick operations allow you to find and select segments within the metafile or commands within the current segment. One pick routine enables highlighting of commands. For a description of the pick routines, see Section 10.5).

Editing commands allow you to insert, delete, and replace commands easily (see Section 10.10). The pick routines enable access to the segments and commands that you want to edit.

The user environment of interactive applications is improved by 2D GMR routines that delete commands from a file, erase the contents of an entire segment, and copy segments (see Section 10.11).

A set of routines enables you to find out the type of command in a file. Each of these inquiring routines is designed to read back the contents of a command from the file and return the values stored in the file, in the form originally used to store that command in the file. For a description of these routines, see Section 10.14 and Appendix C.

9.2. Changing the Picture

Functions:

GM_$MODELCMD_SET_MODE
GM_$MODELCMD_INQ_MODE

You can change the appearance of a picture by changing the commands in the segment that defines the picture. To do this, you use editing routines to change parameters of commands in the segment. The editing functions in 2D GMR include replacing one command with another, replacing one parameter within a command with another, and rubberbanding a command to experiment with its placement. To perform these editing functions, use GM_$MODELCMD_SET_MODE and GM_$MODELCMD_INQ_MODE. These routines supercede GM_$REPLACE_SET_FLAG and GM_$REPLACE_INQ_FLAG. You may still use the GM_$REPLACE... routines to set and replace flags in programs that include these routines. New programs, however, should use the new routines.

The new routines have the following three types and procedures; the rubberbanding mode is a new type without equivalence in the GM_$REPLACE... routines:

GM_$MODELCMD_INSERT
: Insert the current command at the current position after the current command in the currently open segment. This is equivalent to GM_$REPLACE_SET_FLAG (false).

GM_$MODELCMD_REPLACE
: Replace the current command at the current position in the currently open segment. This is equivalent to GM_$REPLACE_SET_FLAG (true).

GM_$MODELCMD_RUBBERBAND
: Temporarily move (rubberband) the current command. The current command is erased, the screen is updated, and the command is redrawn. This action does not change the metafile.

These editing modes change the meaning of the 2D GMR modeling routines. In insert and replace modes, a call to the modeling routines indicates that a command is to be respectively inserted or replaced at the current position in the currently open segment. In rubberbanding mode, calls to the modeling routines update a special internal "rubberbanding command." This command is not contained in any segment, but is treated as if it were contained in the currently open segment. In particular, the coordinates of the command are in the coordinate system of the open segment.

To edit an instanced segment, you must open it. You cannot edit an instanced segment from an instancing segment.

In rubberbanding mode, a call to a modeling routine causes the following three actions to occur:

- The rubberband command is XOR'ed onto the screen. (Thus erasing it.)
- The rubberband command is updated according to the modeling command.
- The rubberband command is XOR'ed onto the screen. (Thus drawing it.)

Rubberband mode and the current insert and replace modes have some important differences.

Rubberbanding makes no permanent change to the screen and makes no change at all to the metafile. Rubberbanding is simply a method by which an application program can interactively get information from the user by means of a pointing device. After getting the information, the application program must then insert or replace the command with the desired values of changes.

The GM_$REFRESH_UPDATE refresh state is often valuable in conjunction with replace mode. For more information on refresh states, see Section 10.2.

9.3. An Interactive Program

The following program creates one segment containing sixteen filled polyline commands. The user can then pick and move the commands. The program illustrates the following:

- The modeling command GM_$POLYLINE_2D16 is used in all three model command modes: gm_$modelcmd_insert, gm_$modelcmd_replace, and gm_$modelcmd_rubberband.
- Partial refresh is used. When you run the program, move one of the polygons over another, then move it again to another location. You will see that partial refresh may not accurately update the viewport (see Section 10.2).
- Command highlighting is used to show which command is picked.
- GM_$PICK_COMMAND returns a picked command only if the command intersects the pick aperture. Note that a nonzero pick aperture makes picking somewhat easier. A nonzero pick aperature is essential when there are horizontal or vertical lines to pick (see Section 10.5).

```
PROGRAM star_move;

{ The following keys are enabled and perform the following actions:

    P ..... Toggle to pick/replace a command
    Q ..... Quit
   ^X ..... Abort rubberbanding (command is NOT replaced)
}

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/gmr.ins.pas';
%INCLUDE '/sys/ins/pfm.ins.pas';
%LIST;

CONST

    ctrlx   = CHR( 16#18 );
    stars_x = 4;
    stars_y = 4;

VAR

    ev_type                     : gm_$event_t;
    character                  : CHAR;
    bitmap_pos                 : gm_$pointreal_t;
    viewport_id                : INTEGER;
    segment_pos                : gm_$pointreal_t;
    status                     : status_$t;

    flush_ev_type              : gm_$event_t;
    flush_character            : CHAR;
    flush_bitmap_pos           : gm_$pointreal_t;
    flush_viewport_id          : INTEGER;
    flush_segment_pos          : gm_$pointreal_t;
    flush_status               : status_$t;

    i                          : INTEGER;
    j                          : INTEGER;
    k                          : INTEGER;

    star                       : gm_$point_array16_t;
    vertices                   : INTEGER;
    closed                     : BOOLEAN;
    filled                     : BOOLEAN;

    last_segment_pos           : gm_$pointreal_t;
    delta                      : gm_$point16_t;

    file_id                    : INTEGER;
    sid                        : gm_$segment_id_t;

    n_instances                : INTEGER32;
    bounds                     : gm_$boundsreal_t;

    command_picked             : BOOLEAN;

    bitmap_size                : gm_$point16_t := [ 1024, 1024 ];
    pick_aperture              : gm_$pointreal_t := [ 4.0, 4.0 ];

    PROCEDURE check;
    BEGIN
        IF status.all <> status_$ok
        THEN pfm_$error_trap( status );
        END;

BEGIN

    gm_$init                             { Initialize the 2D GMR package. }
        ( gm_$direct
        , 1
        , bitmap_size
        , 8
        , status
        );
    check;

    gm_$file_create            { Create a 2D GMR file named 'stars.gmr.' }
        ( 'stars.gmr'
        , SIZEOF( 'stars.gmr' )
        , gm_$overwrite
        , gm_$1w
        , file_id
    , status
    );
check;

gm_$data_coerce_set_real            { Coerce REAL data to INTEGER32. }
    ( gm_$32
    , status
    );
check;

gm_$segment_create                  { Create an unnamed segment. }
    ( ''
    , 0
    , sid
    , status
    );
check;

star[ 1 ].x := 000;                 { Define a filled polyline. }
star[ 1 ].y := 000;
star[ 2 ].x := 400;
star[ 2 ].y := 300;
star[ 3 ].x := 000;
star[ 3 ].y := 300;
star[ 4 ].x := 400;
star[ 4 ].y := 000;
star[ 5 ].x := 200;
star[ 5 ].y := 400;
closed := TRUE;
filled := TRUE;
vertices := 5;

gm_$modelcmd_set_mode               { Set model command mode to insert. }
    ( gm_$modelcmd_insert
    , status
    );
check;

FOR i := 1 TO stars_x               { Insert polylines into the segment. }
DO BEGIN
    FOR j := 1 TO stars_y
    DO BEGIN
        gm_$polyline_2d16
            ( vertices
            , star
            , closed
            , filled
            , status
            );
        check;
        FOR k := 1 TO vertices
        DO star[ k ].y := star[ k ].y + 600;
        END;
    FOR k := 1 TO vertices
    DO BEGIN
        star[ k ].x := star[ k ].x + 600;
        star[ k ].y := star[ k ].y - 600 * stars_y;
        END;
    END;
```

```
gm_$display_segment                { Display the segment. }
    ( sid
    , status
    );
check;

gm_$viewport_set_refresh_state     { Set the refresh state to partial. }
    ( gm_$refresh_partial
    , status
    );
check;

gm_$cursor_set_active              { Make the cursor active. }
    ( TRUE
    , status
    );
check;

gm_$input_enable                   { Enable keys ^X, P, and Q. }
    ( gm_$keystroke
    , [ ctrlx
      , 'P'
      , 'Q'
      , 'p'
      , 'q'
      ]
    , status
    );
check;

gm_$input_enable                   { Enable locator events. }
    ( gm_$locator
    , []
    , status
    );
check;

command_picked := FALSE;

REPEAT

    gm_$input_event_wait           { Wait for an event. }
        ( TRUE
        , ev_type
        , character
        , bitmap_pos
        , viewport_id
        , segment_pos
        , status
        );
    check;

    IF ev_type = gm_$locator       { Flush the queue. }
    THEN REPEAT
        gm_$input_event_wait
            ( FALSE
            , flush_ev_type
            , flush_character
            , flush_bitmap_pos
        , flush_viewport_id
        , flush_segment_pos
        , flush_status
        );
    IF flush_ev_type <> gm_$no_event
    THEN BEGIN
        ev_type     := flush_ev_type;
        character   := flush_character;
        bitmap_pos  := flush_bitmap_pos;
        viewport_id := flush_viewport_id;
        segment_pos := flush_segment_pos;
        status      := flush_status;
        check;
        END;
    UNTIL flush_ev_type <> gm_$locator;

CASE ev_type OF                 { Do case event type. }

    gm_$keystroke:

    CASE character OF

        'P',
        'p':
        IF NOT command_picked
        THEN BEGIN

            gm_$pick_set_center         { Set the pick center. }
                ( segment_pos
                , status
                );
            check;
            gm_$pick_set_size           { Set the pick aperture. }
                ( pick_aperture
                , status
                );
            check;
            gm_$pick_segment            { Clear the old pick list. }
                ( gm_$clear
                , sid
                , n_instances
                , bounds
                , status
                );
            check;
            gm_$pick_segment        { Set up pick at the top segment. }
                ( gm_$setup
                , sid
                , n_instances
                , bounds
                , status
                );
            IF status.all = status_$ok
            THEN BEGIN
                gm_$pick_command      { Initialize the pick_command. }
                    ( gm_$start
                    , status
                    );
                check;
            gm_$pick_command         { Pick a command. }
                ( gm_$cnext
                , status
                );
            command_picked := status.all = status_$ok;
            END;
        IF command_picked
        THEN BEGIN
            gm_$pick_highlight_command   { Highlight the picked }
                ( gm_$outline            { command. }
                , 1.0
                , status
                );
            check;
            gm_$inq_polyline_2d16    { Inquire about the }
                                     { picked polyline. }
                ( vertices
                , star
                , closed
                , filled
                , status
                );
            check;
            gm_$modelcmd_set_mode    { Change to rubberband mode. }
                ( gm_$modelcmd_rubberband
                , status
                );
            check;
            gm_$cursor_set_active    { Turn off the cursor. }
                ( FALSE
                , status
                );
            check;
            last_segment_pos := segment_pos;
            END;
        END

    ELSE BEGIN

        gm_$modelcmd_set_mode        { Change to replace mode. }
            ( gm_$modelcmd_replace
            , status
            );
        check;
        gm_$polyline_2d16            { Replace the polyline. }
            ( vertices
            , star
            , closed
            , filled
            , status
            );
        check;
        gm_$cursor_set_active        { Turn the cursor on. }
            ( TRUE
            , status
            );
        check;
        command_picked := FALSE;
        END;
```

```
            ctrlx:
            BEGIN
                command_picked := FALSE;
                gm_$modelcmd_set_mode       { Turn off rubberband mode. }
                    ( gm_$modelcmd_replace
                    , status
                    );
                check;
                gm_$cursor_set_active       { Turn the cursor on. }
                    ( TRUE
                    , status
                    );
                check;
                END;

            'Q',
            'q':
            EXIT;                           { Quit. }

            END;

        gm_$locator:
        IF command_picked
        THEN BEGIN

            delta.x := ROUND( segment_pos.x - last_segment_pos.x );
            delta.y := ROUND( segment_pos.y - last_segment_pos.y );
            last_segment_pos := segment_pos;
            FOR i := 1 TO vertices
            DO BEGIN
                star[ i ].x := star[ i ].x + delta.x;
                star[ i ].y := star[ i ].y + delta.y;
                END;
            gm_$polyline_2d16               { Move XOR-rubberband. }
                ( vertices
                , star
                , closed
                , filled
                , status
                );
            check;
            END

        ELSE BEGIN

            gm_$cursor_set_position( bitmap_pos, status );
            check;
            END;

        END;

    UNTIL FALSE;

gm_$segment_close                           { Close the segment. }
    ( TRUE
    , status
    );
check;
```

```
gm_$file_close                          { Close the file. }
    ( TRUE
    , status
    );
check;

gm_$terminate                           { Terminate the session. }
    ( status
    );
check;

END.
```

Chapter 10
Routines for Interactive Applications

This chapter describes the following interactive functions: replacing commands, establishing a refresh state, controlling the cursor, using input operations and event reporting, picking operations, editing a metafile, and reading a metafile. Examples illustrate many of these functions.

10.1. Editing Modes

```
Functions:

GM_$MODELCMD_SET_MODE
GM_$MODELCMD_INQ_MODE
GM_$REPLACE_SET_FLAG
GM_$REPLACE_INQ_FLAG
```

The routine GM_$MODELCMD_SET_MODE establishes the editing mode to replace, insert, or move (rubberband) a command (see Section 9.2).

The GM_$MODELCMD... routines supercede the two replace routines described in this section. GM_$REPLACE_SET_FLAG and GM_$REPLACE_INQ_FLAG are still useable in programs that include them. The new routines are recommended for new programs.

GM_$MODELCMD_SET_MODE (gm_$modelcmd_replace,*) puts the 2D GMR package in a mode in which commands are replaced (overwritten) within the file, not inserted. In the replace state, the current command is continually replaced. This is equivalent to GM_$REPLACE_SET_FLAG (true).

In replace mode, you can only replace a command with a command of the same type. Calling a routine that attempts to write any other command type to the file will not affect the file and will reset the mode to GM_$MODELCMD_INSERT.

GM_$MODELCMD_INQ_MODE tells you whether the 2D GMR package is in the (normal) insert, replace, or rubberband mode. GM_$MODELCMD_INQ_MODE relates to GM_$REPLACE_INQ_FLAG in this way:

Mode	Returned Flag	Returned Status
GM_$MODELCMD_INSERT	False	0
GM_$MODELCMD_REPLACE	True	0
GM_$MODELCMD_RUBBERBAND	?	Illegal value

As the list above indicates, you can use the old GM_$REPLACE_FLAG routine for two of the modes, but not with rubberband mode.

10.2. Establishing a Refresh State

```
Functions:

GM_$VIEWPORT_SET_REFRESH_STATE
GM_$VIEWPORT_INQ_REFRESH_STATE
```

GM_$VIEWPORT_SET_REFRESH_STATE allows you to control the frequency at which the display in a viewport is refreshed. This routine allows you to change the metafile and have the package automatically update one or more viewports to incorporate these changes, without calling a refresh routine. One use of this feature is in a rubberbanding procedure when you are trying to find the right place to put a line.

GM_$VIEWPORT_SET_REFRESH_STATE allows you to specify the refresh state of the current viewport. GM_$VIEWPORT_INQ_REFRESH_STATE returns the value of the refresh state of the current viewport. These routines have the following types and procedures:

GM_$REFRESH_INHIBIT
: Changing commands in the file does not immediately affect this viewport. In borrow mode, the viewport is redrawn only when you call GM_$VIEWPORT_REFRESH. In direct mode, the viewport is redrawn only when you call GM_$VIEWPORT_REFRESH, or when the display is refreshed as the result of a DM command that causes the window to be redrawn. Thus, calling GM_$DISPLAY_REFRESH does not affect a viewport in this refresh state.

GM_$REFRESH_WAIT
: (Default) Changing commands in the file does not immediately affect this viewport. In borrow mode, the viewport is redrawn only when you call GM_$VIEWPORT_REFRESH or GM_$DISPLAY_REFRESH. In direct mode, the viewport is redrawn only when you call GM_$VIEWPORT_REFRESH or GM_$DISPLAY_REFRESH or when the display is refreshed as the result of a DM command that causes the window to be redrawn.

GM_$REFRESH_PARTIAL
: Every time you change any command in the file, the following occurs: Inserted primitive commands are added, and deleted primitive commands are erased, but underlying data is not rewritten. This provides faster interactive drawing. You should, however, periodically clean up the accumulating inaccuracies by calling GM_$VIEWPORT_REFRESH to redisplay the viewport.

GM_$REFRESH_UPDATE
: Every time you change any command in the file, this viewport is completely corrected.

Partial refresh has two aspects and uses in applications packages -- partial refresh (erase) and partial refresh (draw):

- Partial refresh (erase) is used when you call GM_$COMMAND_DELETE or when you replace a command to erase the old command. The command is erased by redrawing it in the background color. This may partially erase other commands that overlap the erased command.

- Partial refresh (draw) means drawing a command without regard to precedence. Partial refresh (draw) is used when the application calls a modeling routine while in replace or insert mode.

Partial refresh does not always update the viewport accurately. For completely accurate incremental updating, set the viewport state to GM_$REFRESH_UPDATE. Extensive use of partial refresh may eventually force a call to GM_$VIEWPORT_REFRESH.

10.3. Controlling the Cursor

Functions:

```
GM_$CURSOR_SET_ACTIVE
GM_$CURSOR_SET_PATTERN
GM_$CURSOR_SET_POSITION
GM_$CURSOR_INQ_ACTIVE
GM_$CURSOR_INQ_PATTERN
GM_$CURSOR_INQ_POSITION
```

The cursor is a key element in an interactive program. The 2D GMR package has routines to control cursor activity. position and appearance.

GM_$CURSOR_SET_ACTIVE turns the cursor on and off. Initially, the cursor is off.

GM_$CURSOR_SET_PATTERN establishes a new cursor pattern (up to 16x16 pixels). The cursor pattern is defined as a sequence of rows of bits from top to bottom. Within the cursor pattern, you can specify which pixel is to be considered the cursor origin. You specify the number of pixels that are to be displayed to the left of, and above, the cursor position established by GM_$CURSOR_SET_POSITION (see Figure 10-1).

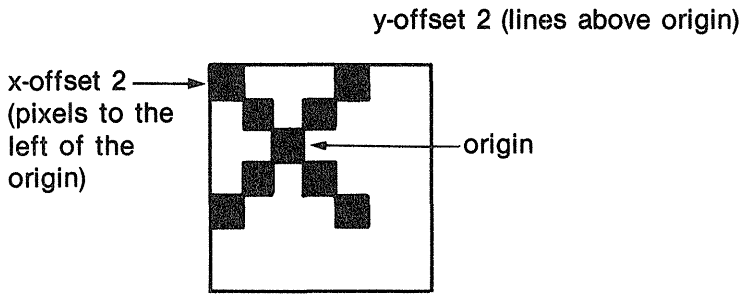

y-offset 2 (lines above origin)
x-offset 2 (pixels to the left of the origin)
origin

Figure 10-1. Cursor Pattern and Position

You must place a cursor pattern smaller than 16x16 in the high-order bits of the first words of the pattern:

```
VAR
{ A cursor pattern smaller than 16x16 starts in the high order bits,
  and starts in word 1 of the array }

cursor_pattern1 : gm_$cursor_pattern_t
                := [16#8080,16#4100,16#2200,16#1400,
                    16#800,16#1400,16#2200,16#4100,16#8080];

cursor_size : gm_$point16_t := [9,9];

cursor_origin : gm_$point16_t := [4,4];
 .
 .
 .

gm_$cursor_set_pattern(gm_$bitmap,cursor_size,
                       cursor_pattern1,cursor_origin, status);
```

GM_$CURSOR_SET_POSITION moves the cursor to a position that you specify as fractions of the size of the GM bitmap.

GM_$CURSOR_INQ... routines return the current values of the cursor parameters.

10.4. Using Input Operations

The 2D GMR package includes a set of routines that enable graphics programs to accept input from various input devices. The input routines can be used to synchronize program execution around input events. These input routines function only in direct and in borrow mode.

In within-GPR mode, you must use GPR input routines (see Chapter 11).

10.4.1. Event Types

Functions:

GM_$INPUT_ENABLE
GM_$INPUT_DISABLE

An event occurs when input is generated in a window (direct mode) or in the borrowed display (borrow mode). The 2D GMR package supports several classes of events, called event types. Programs use an input routine to select the type of event to be reported to them; this operation is called enabling an event type. The event types are the following:

GM_$INPUT_ENABLE enables a single type of input event. To enable multiple input types, call this procedure multiple times. No input events are enabled as a default.

GM_$INPUT_DISABLE disables a single type of input event. To disable multiple input types, call this procedure multiple times.

Table 10-1. Event Types

Keystroke	A keystroke event occurs when you type a keyboard character. Programs can select a subset of keyboard characters, called a keyset, to be recognized as keystroke events. In direct mode, keys that do not belong to the keyset are processed normally by the Display Manager. In borrow mode, keys not belonging to the keyset are ignored.
Button	A button event occurs when you press a button on the mouse or bitpad puck.
Locator	A locator event occurs when you move the mouse or the bitpad puck, or use the touchpad.
Locator stop	A locator stop event occurs when you stop moving the mouse or bitpad puck, or stop using the touchpad.
Window transition event	In direct mode, the cursor may move into and out of the window in which the GM bitmap resides. When the cursor leaves the window, the input routines report to the program an event of type GM_$LEFT_WINDOW; when the cursor enters the window, the routines report an event type of GM_$ENTERED_WINDOW.

10.4.2. Event Reporting

`Function:`

`GM_$INPUT_EVENT_WAIT`

When you enable an event type, the input routines will report each event of the enabled type to the program, along with a cursor position. This position is in bitmap coordinates, that is, in fractions of the GM bitmap.

When you call GM_$INPUT_EVENT_WAIT, you may specify whether the process is to wait until an enabled event occurs or to return a GM_$NO_EVENT event type if no event has occurred. The first argument of the routine controls this choice. (These alternatives are similar to the choice between the routines GPR_$EVENT_WAIT and GPR_$COND_EVENT_WAIT.)

In borrow mode, events that have not been enabled are ignored. In direct mode, all events outside the Display Manager window in which 2D GMR is running are handled by the Display Manager. In addition, events that have not been enabled are passed to the Display Manager.

If the enabled event type is keystroke or button, the input routines return an ASCII character from the enabled keyset. When defining a keyset for a keystroke event, consult the system insert files /SYS/INS/KBD.INS.PAS, /SYS/INS/KBD.INS.FTN, and /SYS/INS/KBD.INS.C. These files contain the definitions for the non-ASCII keyboard keys in the range 128 through 255. For a sample keyboard chart, see Appendix B.

The input routines report button events as ASCII characters. "Down" transitions range from "a" to "d"; "up" transitions range from "A" to "D". The three mouse keys start with (a/A) on the left side. As with keystroke events, button events can be selectively enabled by specifying a button keyset.

Locator events report coordinates of the locator input, expressed in fraction-of-GM-bitmap coordinates. If the program has not enabled locator events, then at the next occurrence of an enabled event, the 2D GMR software reports the locator final cursor position to the program, along with the enabled event.

10.5. Using Picking

Pick operations allow you to find and select segments within the metafile or to find and select commands within the current picked segment. Picking may be done with or without respect to a pick aperture.

10.5.1. Picking Without an Aperture

When you perform picking without an aperture, you use only GM_$PICK_COMMAND; and you may only use the options GM_$STEP, GM_$START, and GM_$END. These options to GM_$PICK_COMMAND allow you to move about in the currently open segment and to change which command is the current command, that is, the command before which new modeling commands will be inserted.

10.5.2. Picking With an Aperture

The search for segments or commands can be limited to a specified range of coordinates. This range is called the pick aperture. Before starting to pick, you can define the pick aperture using the routines GM_$SET_PICK_CENTER and GM_$SET_PICK_SIZE. The coordinates and size used to set the pick aperture are in the coordinate system of the top-level segment in the viewport, that is, the coordinate system which GM_$INPUT_EVENT_WAIT uses to report locator events. The pick aperture is initialized to center (0,0) and size (0,0).

Picking with an aperture requires two steps.

1. Use GM_$PICK_SEGMENT to define an instance path to the desired segment.

2. Using this instance path, you may use GM_$PICK_COMMAND to pick individual commands within the picked segment.

Because a given segment may be instanced many times, an instance path to the segment is necessary to specify which instance is intended. Consider the following example. Suppose segment A is a stick man which is instanced twice into segment B and viewed in the viewport as shown in Figure 10-2:

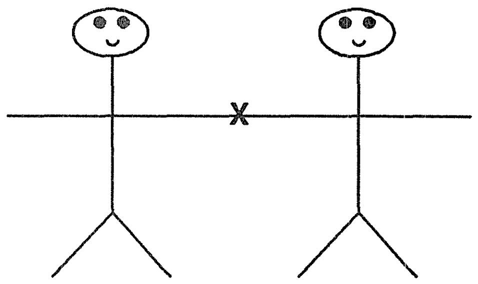


Figure 10-2. Instancing and Picking

The "X" is the pick center. Note that without specifying an instance path, that is, without specifying which instance is to be considered, the pick is ambiguous. It is unclear whether the left arm or the right arm has been selected.

When picking with an aperture, use the following general sequence of operations:

1. Close any open segment. (This is not always necessary, but you must do it before step 5.)

2. Call GM_$PICK_SEGMENT (gm_$setup, *) to initialize segment picking.

3. Call GM_$PICK_SET_CENTER and GM_$PICK_SET_SIZE to define the pick aperture.

4. Call GM_$PICK_SEGMENT (appropriate option, for example, gm_$down, *)

5. Determine whether a segment was picked, and if so, whether it is of interest. If a segment was not picked or the picked segment is not of interest, go back to step 4 or step 3.

6. Open the picked segment.

7. Call GM_$PICK_COMMAND (gm_$start, *) to initialize command picking.

8. Call GM_$PICK_COMMAND (gm_$cnext or gm_$step, *)

9. Determine whether a command was picked, and if so, whether it is of interest. If a command was not picked or the picked command is not of interest, go back to step 8; or if you are at the end of the segment, close the segment and go back to step 4 or step 3.

10. Edit the picked and open segment. If you want to edit interactively using a locator

device, then use GM_$PICK_TRANSFORM_POINT to convert coordinates from the top-level segment, that is, as reported by GM_$INPUT_EVENT_WAIT, to the coordinate system of the open segment.

11. Close the segment when you have completed editing.

10.6. Setting the Pick Aperture

Functions:

```
GM_$PICK_SET_CENTER
GM_$PICK_SET_SIZE
GM_$PICK_INQ_CENTER
GM_$PICK_INQ_SIZE
```

Setting the center and size of the pick aperture defines the part of coordinate space to use in searches. The size of the pick aperture may be large, small, or zero. GM_$PICK_SET_CENTER sets the center of the pick aperture in segment coordinates. GM_$PICK_INQ_CENTER returns the center of the pick aperture.

GM_$PICK_SET_SIZE sets the x and y tolerances for the pick aperture, in segment coordinates. GM_$PICK_INQ_SIZE returns the x and y tolerances for the pick aperture.

10.7. Picking and Listing Segments

Functions:

```
GM_$PICK_SEGMENT
GM_$PICK_INQ_LIST
GM_$PICK_HIGHLIGHT_SEGMENT
GM_$PICK_TRANSFORM_POINT
```

GM_$PICK_SEGMENT looks for segments within the pick aperture. The search for segments follows one of eight rules shown in Table 10-2:

If a segment is found, it becomes the current picked segment and the last segment on the list of picked segments. If no segment is found, the list of picked segments is unchanged.

While a segment is in the list of picked segments, you may not delete or edit it. A picked segment must be open before you can pick commands in it. Therefore, if the open segment is not the picked segment, you must close it. You can then open the picked segment to make it the current segment.

GM_$PICK_INQ_LIST returns the current list of picked segments. The first segment in the list is the segment in which the segment picking process was initialized. For example, assume that your file contains segments that instance other segments, as shown in Figure 10-3:

Table 10-2. Search Rules for Picking

GM_$SETUP	Make the viewport primary segment of the current viewport the start of the list of picked segments. The rest of the list is emptied.
GM_$DOWN	Find the first segment within the current picked segment that, when instanced, falls within the pick aperture.
GM_$NEXT	Find the next segment within the segment one higher in the list of picked segments, that falls within the pick aperture.
GM_$UP	Move up one level in the list of picked segments.
GM_$TOP	Proceed to the top segment in the list of picked segments, destroying the rest of the list of picked segments.
GM_$CLEAR	Clear the entire list of picked segments, allowing all segments to be edited or deleted.
GM_$BOTTOM	Perform GM_$DOWN repeatedly until a segment is reached for which GM_$DOWN finds nothing.
GM_$NEXTBOTTOM	Perform GM_$BOTTOM. If nothing is found, perform GM_$NEXT, or one or more GM_$UP's followed by a GM_$NEXT, until a GM_$NEXT finds a segment. When a GM_$NEXT finds a segment, perform a GM_$BOTTOM from there.

```
        1
       / \
      2   3
     / \
    4   5
   /   /
  6   8
 /
7
```

Figure 10-3. Instancing and Picking Segments

If segment 1 is the viewport primary segment of the current viewport, then calling GM_$PICK_SEGMENT with the following sequence of options will change the current picked segment and the list of picked segments, as indicated in Table 10-3.

To find all lowest-level segments in the pick aperature, one at a time use the GM_$SETUP option, then GM_$BOTTOM, then GM_$NEXTBOTTOM until no further segments are found.

You can generate a list of pickable segments directly instanced by another segment by picking that segment, then using GM_$DOWN, then using GM_$NEXT repeatedly until no more matches are found.

Table 10-3. **Example of Picking and Listing Segments**

Option	New Current Picked Segment	List of Picked Segments
GM_$SETUP	1	1
GM_$DOWN	2	1,2
GM_$DOWN	4	1,2,4
GM_$NEXT	5	1,2,5
GM_$UP	2	1,2
GM_$BOTTOM	7	1,2,4,6,7
GM_$NEXTBOTTOM	8	1,2,5,8
GM_$NEXTBOTTOM	3	1,3
GM_$TOP	1	1
GM_$CLEAR	none	none

GM_$PICK_HIGHLIGHT_SEGMENT highlights the picked segment on the display. Only one instance of a segment is highlighted.

GM_$PICK_TRANSFORM_POINT allows you to convert viewport segment coordinates to the segment coordinates of an instance of the picked segment. While any segment may be edited, GM_$INPUT_EVENT_WAIT still reports the locator position in viewport (that is, top level) segment coordinates. This presents no problem if only translations are involved. However, if scaling or rotation is used, particularly with multilevel instancing, then going from the viewport segment coordinates to the open segment coordinates is more difficult. This call facilitates that conversion.

10.8. Picking a Command

Functions:

GM_$PICK_COMMAND
GM_$PICK_HIGHLIGHT_COMMAND

Pick routines let you select a single entity from a file, either a segment or a command. As you edit a segment, you can use the pick routines to select the command you want to change.

In editing a file, you may want to change some segments of it and protect others from change. You can do this by using pick mask and pick threshold routines to protect a basic picture while you change some elements in it.

You may only edit commands within the current segment. The current segment has a current command. When you open or create a segment, that segment becomes the current segment, and the last command in the current segment becomes the current command. You can append new commands to this current command.

The routine GM_$PICK_COMMAND can change the current command. When the pick-command procedure finds a command, it becomes the current command. You can then read or edit that command.

When you insert a new command into the metafile, it is placed just after the current command, and it becomes the current command. Thus, if you follow an insertion with a deletion, the last command you inserted is deleted. For a description of editing procedures, see Section 10.10.

GM_$PICK_COMMAND looks for commands within the current segment by following one of four search rules shown in Table 10-4:

Table 10-4. Search Rules for Picking a Command

GM_$CNEXT	(Next command) Find the next command which falls within the pick aperture, moving forward in the segment.
GM_$STEP	(Forward one step) Find the next command in the segment, independent of the pick aperture.
GM_$START	Make the current command the blank space at the start of the segment. This allows a search to proceed from the start of the segment, or it allows commands to be inserted at the start of the segment.
GM_$END	Make the final command in the segment the current command. This allows you to insert additional commands at the end of the segment.

You may want to search for the next command within a viewport. You can use GM_$CNEXT to find the next command in a segment that falls within the pick aperture. The search may be in a segment that contains commands for several graphic images. The searching process, however, is for commands that fall within the pick aperture.

Instance commands are treated like any other commands in this context. To pick "into" an instanced segment, use GM_$PICK_SEGMENT.

GM_$PICK_HIGHLIGHT_COMMAND highlights the current command on the display.

10.9. Controlling What is Picked

Functions:

```
GM_$PICK_SET_THRESHOLD
GM_$PICK_INQ_THRESHOLD
GM_$PICK_SET_MASK
GM_$PICK_INQ_MASK
```

You can establish the criterion for picking and not picking segments in two ways: by using a threshold to eliminate segments with pickable values too small; or by using a mask to eliminate segments missing certain bits in their pickable values.

To assign pickable values to segments, use GM_$SEGMENT_SET_PICKABLE (see Section 7.2).

A segment is picked only if it meets the pick mask and threshold criteria and the pick aperture criterion. The pick mask criterion requires that at least one bit be "1" in both the segment's pickable value and the pick mask. The pick threshold criterion requires that the segment's pickable value be greater than the pick threshold.

A use for this pair of methods is to give segments containing text even pickable values and segments without text odd values, and to give large segments large pickable values. Then you can use the pick mask to make text segments nonpickable (pick mask = 1), and the pick threshold to avoid searching for small objects.

GM_$PICK_SET_THRESHOLD establishes a minimum segment pickable value. Segments with smaller pickable values are ignored in searches. The default value is 1, in which case all segments of nonzero pickable values are pickable.

GM_$PICK_SET_MASK allows you to base segment pickability on individual bits within the pickable value for each segment. If any bit is 1 in both the mask and the segment pickable number, the segment is pickable. The default value is 16#7FFFFFFF, in which case all segments of nonzero pickable value are pickable.

GM_$PICK_INQ_THRESHOLD and GM_$PICK_INQ_MASK enable you to ascertain the current value of these parameters.

10.10. Editing Metafiles

The 2D GMR package includes commands for efficient editing of files. These commands allow you to insert, delete, and replace commands easily. The pick routines described in this chapter give ready access to the segments and commands that you want to edit.

When you open a segment, the last command in the segment becomes the current command, allowing new commands to be appended. You may use GM_$PICK_COMMAND to change the current command. You can then insert new commands at that point in the segment, or you can then use the editing commands to read, delete, or replace the command.

Using the pick and editing commands is analogous to using editing commands in a text editor. In editing text, you move the cursor to just after a character that you want to change. You then backspace over the item you want to delete. Similarly, you use the command GM_$PICK_COMMAND to make a particular command the current command. You can then use GM_$COMMAND_DELETE to delete the command. Thus, picking a command and then calling GM_$COMMAND_DELETE deletes the command you just picked.

10.11. Deleting and Copying

The routines for deleting and copying facilitate the use of interactive applications by making it easy to manipulate the contents of segments and to copy files.

10.11.1. Deleting

Function:

GM_$COMMAND_DELETE
GM_$SEGMENT_ERASE

GM_$COMMAND_DELETE deletes the current command. After the current command is deleted, the previous command in the current segment becomes the current command. If the first command in the segment is deleted, the blank space at the start of the segment becomes the current command. You must then change the current command before deleting any more commands.

GM_$SEGMENT_ERASE deletes all commands in the current segment and leaves the segment open so that you may insert new commands.

10.11.2. Copying

Function:

GM_$SEGMENT_COPY

GM_$SEGMENT_COPY copies the entire contents of another segment into the current segment.

Note the difference between GM_$SEGMENT_COPY and the GM_$INSTANCE... routines. GM_$SEGMENT_COPY leaves you with two copies of the segment, allowing you to modify the two copies separately. The GM_$INSTANCE... routines leave you with one copy of the segment and a reference to that segment, so that all displayed instances can be changed by modifying the single instanced segment.

10.12. Program with Picking

The program in this section illustrates the use of all the pick functions and shows the required order of their use. You can use this program with any metafile, for example with the metafile "hotel.gm" created by the hotel program in Appendix D.

The listing of insert files at the top of the program includes gpr.ins.pas. This file gives access to the DOMAIN Graphics Primitives (GPR) routines. In general, the mixing of 2D GMR and GPR is not recommended unless you specify within-GPR mode with the routine GM_$INIT. Here the GPR routine provides the best way to release the display that 2D GMR must acquire. The GPR routine releases the display to allow writing output to a stream.

In addition, the program illustrates creating two windows: one as a transcript pad for text and one for display of graphics. You create an extra window with pad_$create. The command returns a stream id which you then use as the unit parameter in GM_$INIT.

The routine GM_$INPUT_ENABLE provides translation for function keys to create an easy user interface.

The program pick.pas illustrates the use of the various options to GM_$PICK_SEGMENT and GM_$PICK_COMMAND. The program prompts the user for a metafile pathname, creates a pad in which to display the metafile, then allows the user to make calls to GM_$PICK_SEGMENT and GM_$PICK_COMMAND. Status information from the calls is echoed in the original window.

The following keyboard map is used:

```
        key              action taken by program

        C, c             GM_$SEGMENT_CLOSE( picked_segment, *
        H, h             GM_$PICK_HIGHLIGHT_SEGMENT( picked_segment, *
                         GM_$PICK_HIGHLIGHT_COMMAND( picked_command, *
        O, o             GM_$SEGMENT_OPEN( picked sement, *
        P, p             GM_$PICK_SET_CENTER( current mouse position, *
                         GM_$PICK_SET_SIZE( 4x4, *
        Q, q             quit

        F1               GM_$PICK_SEGMENT( gm_$clear, *
        F2               GM_$PICK_SEGMENT( gm_$setup, *
        F3               GM_$PICK_SEGMENT( gm_$top, *
        F4               GM_$PICK_SEGMENT( gm_$next, *
        F5               GM_$PICK_SEGMENT( gm_$up, *
        F6               GM_$PICK_SEGMENT( gm_$down, *
        F7               GM_$PICK_SEGMENT( gm_$bottom, *
        F8               GM_$PICK_SEGMENT( gm_$nextbottom, *

        F1 shifted       GM_$PICK_COMMAND( gm_$cnext, *
        F2 shifted       GM_$PICK_COMMAND( gm_$step, *
        F3 shifted       GM_$PICK_COMMAND( gm_$start, *
        F4 shifted       GM_$PICK_COMMAND( gm_$end, *
```

```
 PROGRAM pick;


%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pfm.ins.pas';
%INCLUDE '/sys/ins/pad.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/gpr.ins.pas';        { <- !!! }
%INCLUDE '/sys/ins/gmr.ins.pas';
%LIST;

CONST
```

```
    cmd_cnext       = CHAR( 16#D0 );    { Next command inside pick aperture }
    cmd_step        = CHAR( 16#D1 );    { Forward one command in segment );
                                          ignore pick aperture }
    cmd_start       = CHAR( 16#D2 );    { Backward to before first command in
                                          segment ); ignore pick aperture }
    cmd_end         = CHAR( 16#D3 );    { Forward to last command in segment );
                                          ignore pick aperture }

    seg_clear       = CHAR( 16#C0 );
    seg_setup       = CHAR( 16#C1 );
    seg_top         = CHAR( 16#C2 );
    seg_next        = CHAR( 16#C3 );    { Next occurrence at this level inside
                                          pick aperture }
    seg_up          = CHAR( 16#C4 );    { Next higher level in segment
                                          hierarchy }
    seg_down        = CHAR( 16#C5 );    { Next lower level segment inside
                                          pick aperture }
    seg_bottom      = CHAR( 16#C6 );    { Lowest level segment inside pick
                                          aperture }
    seg_nextbottom  = CHAR( 16#C7 );    { Lowest level segment inside pick
                                          aperture }

VAR

    show_window               : pad_$window_desc_t;
    strid                     : stream_$id_t;

    name                      : name_$pname_t;
    length                    : INTEGER;
    size                      : gm_$point16_t := [ 1024, 1024 ];
    file_id                   : INTEGER;
    sid                       : gm_$segment_id_t;

    ev_type                   : gm_$event_t;
    key                       : CHAR;
    bitmap_pos                : gm_$pointreal_t;
    viewport_id               : INTEGER;
    segment_pos               : gm_$pointreal_t;
    status                    : status_$t;

    flush_ev_type             : gm_$event_t;
    flush_key                 : CHAR;
    flush_bitmap_pos          : gm_$pointreal_t;
    flush_viewport_id         : INTEGER;
    flush_segment_pos         : gm_$pointreal_t;
    flush_status              : status_$t;

    pick_box                  : gm_$pointreal_t;
    seg_id                    : gm_$segment_id_t;
    ninstances                : INTEGER32;
    bounds                    : gm_$boundsreal_t;

    search_command            : gm_$search_command_t;
    search_segment            : gm_$search_segment_t;

    pick_command_name         : ARRAY [ gm_$search_command_t ] OF string :=
                                      [ 'PICK_CMD_NEXT%'
                                      , 'PICK_CMD_STEP%'
                                      , 'PICK_CMD_START%'
```

```
                                            , 'PICK_CMD_END%'
                                            ];

    pick_segment_name        : ARRAY [ gm_$search_segment_t ] OF string :=
                                            [ 'PICK_SEG_CLEAR%'
                                            , 'PICK_SEG_SETUP%'
                                            , 'PICK_SEG_TOP%'
                                            , 'PICK_SEG_NEXT%'
                                            , 'PICK_SEG_UP%'
                                            , 'PICK_SEG_DOWN%'
                                            , 'PICK_SEG_BOTTOM%'
                                            , 'PICK_SEG_NEXT_BOT%'
                                            ];

    PROCEDURE squawk
        ( IN  status     : UNIV status_$t
        ; IN  key        : UNIV error_$string_t
        ; IN  str        : UNIV error_$string_t
        ; IN  a1,a2,a3,a4,a5,a6,a7,a8,a9,a10:  UNIV error_$integer32
        ); OPTIONS( VARIABLE );

    VAR

        i            : INTEGER;
        acq_rel_cnt  : INTEGER;
        unobscured   : BOOLEAN;
        st           : status_$t;
        keylen       : INTEGER;

    BEGIN

        keylen := 1;
        WHILE ( key[ keylen ] <> '%' ) AND ( keylen < SIZEOF( key ) )
        DO keylen := keylen + 1;

        gpr_$force_release( acq_rel_cnt, st );

        error_$print_format( status, stream_$stdout, 'I', key, keylen-1,
                             str, a1,a2,a3,a4,a5,a6,a7,a8,a9,a10 );

        FOR i := acq_rel_cnt - 1 DOWNTO 0
        DO unobscured := gpr_$acquire_display( st );

        END;

BEGIN

    WRITE( 'File name:  ' );
    READLN( name );

    length := LASTOF( name );
    WHILE ( name[ length ] = ' ' ) AND ( length > 0 )
    DO length := length - 1;

    show_window.top    := 0;
    show_window.left   := 0;
    show_window.width  := 1023;
    show_window.height := 600;
```

```
pad_$create_window( '', 0, pad_$transcript, 1, show_window, strid,
                   status);
IF status.all <> status_$ok THEN pfm_$error_trap( status );

pad_$set_auto_close( strid, 1, TRUE, status );
IF status.all <> status_$ok THEN pfm_$error_trap( status );

pad_$set_scale( strid, 1, 1, status );
IF status.all <> status_$ok THEN pfm_$error_trap( status );

gm_$init( gm_$direct, strid, size, 8, status );
IF status.all <> status_$ok THEN pfm_$error_trap( status );

gm_$file_open( name, length, gm_$wr, gm_$1w, file_id, status );
IF status.all <> status_$ok THEN pfm_$error_trap( status );

gm_$display_file( status );
IF status.all <> status_$ok THEN pfm_$error_trap( status );

gm_$cursor_set_active( TRUE, status );
IF status.all <> status_$ok THEN pfm_$error_trap( status );

gm_$input_enable
    ( gm_$keystroke
    , [ 'C', 'H', 'O', 'P', 'Q', 'c', 'h', 'o', 'p', 'q'
      , cmd_cnext, cmd_step, cmd_start, cmd_end
      , seg_clear, seg_setup, seg_top, seg_next
      , seg_up, seg_down, seg_bottom, seg_nextbottom
      ]
    , status
    );
IF status.all <> status_$ok THEN pfm_$error_trap( status );

gm_$input_enable( gm_$locator, [], status );
IF status.all <> status_$ok THEN pfm_$error_trap( status );

INTEGER32( seg_id ) := -1;

REPEAT

    gm_$input_event_wait( TRUE, ev_type, key, bitmap_pos, viewport_id,
                          segment_pos, status );
    IF status.all <> status_$ok THEN pfm_$error_trap( status );

    IF ev_type = gm_$locator
    THEN REPEAT

        gm_$input_event_wait( FALSE, flush_ev_type, flush_key,
                              flush_bitmap_pos, flush_viewport_id,
                              flush_segment_pos, flush_status );

        IF flush_ev_type <> gm_$no_event
        THEN BEGIN
            ev_type     := flush_ev_type;
            key    := flush_key;
            bitmap_pos  := flush_bitmap_pos;
            viewport_id := flush_viewport_id;
            segment_pos := flush_segment_pos;
            status      := flush_status;
            IF status.all <> status_$ok THEN pfm_$error_trap( status );
            END;

        UNTIL flush_ev_type <> gm_$locator;

    IF ev_type = gm_$locator
    THEN BEGIN

        gm_$cursor_set_position( bitmap_pos, status );
        IF status.all <> status_$ok THEN pfm_$error_trap( status );

        END

    ELSE IF ev_type = gm_$keystroke
    THEN CASE key OF

        'C','c':
        BEGIN

            gm_$segment_close( FALSE, status );

            squawk( status, 'SEGMENT_CLOSE%', 'Segment id = %8ZLH%$',
                    seg_id );

            IF status.all = status_$ok
            THEN INTEGER32( seg_id ) := -1;

            END;

        'H','h':
        BEGIN

            gm_$pick_highlight_segment( gm_$outline, 1.0, status );
            gm_$pick_highlight_command( gm_$outline, 1.0, status );

            END;

        'O','o':
        BEGIN

            gm_$segment_open( seg_id, status );
            squawk( status, 'SEGMENT_OPEN%','Segment id = %8ZLH%$',
                    seg_id );

            END;

        'P','p':
        BEGIN

            gm_$pick_set_center( segment_pos, status );

            squawk( status, 'PICK_SET_CENTER%', 'Center = ( %WF, %WF )%$',
                    segment_pos.x, segment_pos.y );

            IF status.all = 0
            THEN BEGIN

                pick_box.x := 4.0;
                pick_box.y := 4.0;
            gm_$pick_set_size( pick_box, status );
            squawk( status, 'PICK_SET_SIZE%', 'Size = ( %WF, %WF )%$',
                    pick_box.x, pick_box.y );

            END;

        END;

    'Q','q':
    EXIT;

    cmd_cnext, cmd_step, cmd_start, cmd_end:
    BEGIN

        search_command := gm_$search_command_t( ORD( key ) -
                                                ORD( cmd_cnext ) );

        gm_$pick_command( search_command, status );

        squawk( status, pick_command_name[ search_command ], '%$' );

        IF status.all = status_$ok
        THEN gm_$pick_highlight_command( gm_$outline, 1.0, status );

        END;

    seg_clear, seg_setup, seg_top, seg_next, seg_up, seg_down,
    seg_bottom, seg_nextbottom:
    BEGIN

        search_segment := gm_$search_segment_t( ORD( key ) -
                                                ORD( seg_clear ) );

        gm_$pick_segment( search_segment, seg_id, ninstances, bounds,
                          status );

        IF      status.all <> status_$ok
        OR ELSE search_segment = gm_$clear
        OR ELSE search_segment = gm_$setup
        THEN squawk(status, pick_segment_name[ search_segment ], '%$')
        ELSE BEGIN

            squawk( status, pick_segment_name[ search_segment ],
                    'SEGMENT ID = %8ZLH  NINSTANCES = %8LD%$',
                     seg_id, ninstances );

            gm_$pick_highlight_segment( gm_$outline, 1.0, status );

            END

        END;

    END;

  UNTIL FALSE;

gm_$terminate
    ( status
    );
```

```
IF status.all <> status_$ok THEN pfm_$error_trap( status );

END.
```

10.13. Program Technique: Locator Events and Cursor Tracking

Processing a long queue of input events can slow the tracking of the cursor. The following program fragment provides a technique for disposing of the queue by reading an input event, taking action on that event, and then emptying the input queue of further locator data.

This technique is especially useful for dragging and rubberbanding an image. In this type of operation, locator data is likely to come in faster than it can be processed, but a redraw should only be done at the most recent cursor position.

```
GM_$INPUT_EVENT_WAIT        { Wait for an event. }
    ( TRUE
    , ev_type
    , character
    , bitmap_pos
    , viewport_id
    , segment_pos
    , status
    );
check;

IF ev_type = gm_$locator    { Flush the queue to make sure you have
                              the most recent locator event. }
THEN REPEAT
    GM_$INPUT_EVENT_WAIT
        ( FALSE
        , flush_ev_type
        , flush_character
        , flush_bitmap_pos
        , flush_viewport_id
        , flush_segment_pos
        , flush_status
        );
    IF flush_ev_type <> gm_$no_event
    THEN BEGIN
        ev_type     := flush_ev_type;
        character   := flush_character;
        bitmap_pos  := flush_bitmap_pos;
        viewport_id := flush_viewport_id;
        segment_pos := flush_segment_pos;
        status      := flush_status;
        check;
        END;
    UNTIL flush_ev_type <> gm_$locator;
```

10.14. Reading Commands

Functions:

```
GM_$INQ_ACLASS
GM_$INQ_CIRCLE_[16,32,REAL]
GM_$INQ_COMMAND_TYPE
GM_$INQ_CURVE_2D[16,32,REAL]
GM_$INQ_DRAW_RASTER_OP
GM_$INQ_DRAW_STYLE
GM_$INQ_DRAW_VALUE
GM_$INQ_FILL_BACKGROUND_VALUE
GM_$INQ_FILL_PATTERN
GM_$INQ_FILL_VALUE
GM_$INQ_FONT_FAMILY
GM_$INQ_INSTANCE_SCALE_2D[16,32,REAL]
GM_$INQ_INSTANCE_TRANSFORM_2D[16,32,REAL]
GM_$INQ_INSTANCE_TRANSLATE_2D[16,32,REAL]
GM_$INQ_PLANE_MASK
GM_$INQ_POLYLINE_2D[16,32,REAL]
GM_$INQ_PRIMITIVE_2D[16,32,REAL]
GM_$INQ_RECTANGLE_[16,32,REAL]
GM_$INQ_TAG
GM_$INQ_TEXT_2D[16,32,REAL]
GM_$INQ_TEXT_BACKGROUND_VALUE
GM_$INQ_TEXT_SIZE
GM_$INQ_TEXT_VALUE
```

Each modeling routine that puts a command into the file has a corresponding inquiring routine. Each of these inquiring routines is designed to read back the contents of a command from the file and return the values stored in the file, in the form originally used to store that command in the file. The GM_$INQ... routine has the same syntax and parameters as the routine used to insert the command into the file, except that some of the parameters are output parameters rather than input parameters.

To inquire about a command, you must use GM_$PICK_COMMAND to make that command the current command. Then use GM_$INQ_COMMAND_TYPE to determine the type of command and the data storage type. You can then call the appropriate GM_$INQ... routine to read the parameters of this particular command.

You must read the command using an appropriate data type. That is, you must use 16-bit inquire routines to read data stored in 16-bit storage format. You must use 32-bit integer or real inquire routines to read data stored in 32-bit storage format.

See Appendix C for an example of a program that prints out the entire contents of a metafile in a form you can read.

Chapter 11
Using Within-GPR Mode

This chapter explains how to extend the 2D GMR package to include GPR routines and user-defined primitives. This extension to the 2D GMR package is illustrated with a sample program.

11.1. Extending the 2D GMR Package

The 2D GMR package has four modes of display that use only 2D GMR routines and commands. In addition, the 2D GMR package has a mode that allows you to extend the package to use a bitmap of the graphics primitives (GPR) package. This mode requires the use of GPR routines within the 2D GMR environment.

11.1.1. Borrow, Direct, and Main-Bitmap Modes

In borrow, direct, and main-bitmap modes, the 2D GMR package produces graphics output in the GM bitmap (the screen, Display Manager window, or main-memory bitmap established when the 2D GMR package was initialized). You can see graphics output or other processes through viewports, which are part or all of the GM bitmap. The view is the picture that you can see in a viewport. Moving or scaling a view moves or scales what you see through the viewport.

When you initialize the 2D GMR package, the command GM_$INIT establishes a single viewport that fills the GM bitmap. You may want to change the size of the viewport or create additional viewports.

You can divide the GM bitmap into multiple viewports. You can specify that you want parts of the metafile displayed and moved independently in separate viewports.

11.1.2. Within-GPR Mode

When you use within-GPR mode, the 2D GMR package produces graphics output in the current GPR bitmap. When using 2D GMR in this mode, you may use the following display routines:

- The four routines that set and inquire about criteria for segment visibility (Section 8.4).
- The routine GM_$DISPLAY_SEGMENT_GPR_2D.
- The two routines linking attribute blocks to the display (Section 13.9.1).

A program that uses within-GPR mode must first initialize the graphics primitives package before it initializes the 2D GMR package. In within-GPR mode, you retain control of the display. This means that you must layout viewports or other types of user interface using GPR viewing routines. In this mode, most of the 2D GMR viewing routines are not available.

In within-GPR mode is an application based on the graphics primitive package. Therefore, all coordinates established in this mode are device-dependent and use the standard graphics primitives coordinates. When you call a viewing routine to display a segment in a particular part of the GPR bitmap, you must specify the transformation from world coordinates (that is, segment coordinates) to bitmap-pixel coordinates. Two routines are available to help you make the conversion from world, or segment, coordinates to bitmap-pixel coordinates: GM_$COORD_SEG_TO_PIXEL_2D and GM_$COORD_PIXEL_TO_SEG_2D.

The key command for displaying in within-GPR mode is GM_$DISPLAY_SEGMENT_GPR_2D. You can only call this routine from within-GPR mode. This command causes a specified segment to be displayed with the specified transformation. The segment and the transformation are specified as parameters of this command.

The segment is displayed into the current GPR bitmap. This bitmap is the one that was current at the time the 2D GMR package was initialized. For the attributes required for display, the current bitmap uses the attribute block that is current at the time of display. These attributes include the clipping window and the plane mask, which must be supplied by this GPR attribute block. The 2D GMR package does not control placement of graphics in within-GPR mode. The package may, in fact, write to the entire DM window unless you have established a clipping window and a plane mask.

GM_$DISPLAY_SEGMENT_GPR_2D, unlike the display routines used in borrow, direct, and main-memory display routines, does not clear the display before drawing. If you want the display cleared, you must call GPR_$CLEAR.

When you use within-GPR mode, you can use 2D GMR attribute commands, attribute class commands, and attribute blocks. The current GPR attribute block specifies the starting values of these attributes. If any attributes are changed as the display process performs the commands in the metafile, the returned state of the GPR attribute block is undefined. Because the 2D GMR package does not necessarily return attributes to their original state, you will probably want to copy a separate GPR attribute block before calling GM_$DISPLAY_SEGMENT_GPR_2D.

The following summarizes what routines are available and not available in within-GPR mode.

Graphics metafile routines available in within-GPR mode:

- Modeling routines, including editing routines.
- Attribute routines, including GM_$ABLOCK_ASSIGN_DISPLAY.
- Display routine GM_$DISPLAY_SEGMENT_GPR_2D; no other display routine.
- Segment visibility criteria routines.
- Picking routines.
- Output routines for creating hard copy.

Graphics metafile routines unavailable in within-GPR mode:

- Viewport routines and viewing routines.
- Refresh routines. (2D GMR does not track data).
- Color map routines. (Use GPR routines.)

- Cursor routines. (Use GPR routines.)
- Input routines. (Use GPR routines.)
- Set refresh procedure. (Use GPR routines.)

11.2. Displaying User-Defined Primitives

Function:

GM_$PRIMITIVE_DISPLAY_2D

GM_$PRIMITIVE_DISPLAY_2D assigns the specified user-defined routine to the specified user-defined primitive number. This causes (GM_$PRIMITIVE_2D) commands using this user-defined primitive number to be displayed (at display time) using this routine.

When your routine is called during the display operation, it is passed an array of transformed points (in screen coordinates) and an array of untransformed parameters. Your routine must use only GPR primitive commands.

The following program fragments define a user-defined primitive and the insert commands that use the primitive.

```
  {  This procedure defines the operation to be performed
     when displaying. }

PROCEDURE WIDE_LINE(IN n_points : integer;
                    IN points : UNIV gm_$point_array16_t;
                    IN n_param : integer;
                    IN param : UNIV gm_$arrayreal_t;
                    OUT st : status_$t);
VAR
   k : integer;
   width : integer;

BEGIN
width := trunc(param[1]);

gpr_$move(points[1].x-width, points[1].y-width, st);
for k := 2 to n_points do
   gpr_$line(points[k].x-width, points[k].y-width, st);

gpr_$move(points[1].x-width, points[1].y+width, st);
for k := 2 to n_points do
   gpr_$line(points[k].x-width, points[k].y+width, st);

gpr_$move(points[1].x+width, points[1].y-width, st);
for k := 2 to n_points do
   gpr_$line(points[k].x+width, points[k].y-width, st);

gpr_$move(points[1].x+width, points[1].y+width, st);
for k := 2 to n_points do
   gpr_$line(points[k].x+width, points[k].y+width, st);
```

```
END;

{ Main program }

VAR
   prim_ptr : gm_$primitive_ptr_t;
   .
   .
   .

         { Connect this procedure with this primitive type. }

prim_ptr := addr(wide_line);
gm_$primitive_display_2d(wide_line_type,prim_ptr,st);
   .
   .
   .

         { Insert a command to use this primitive type. }

gm_$primitive_2d16(wide_line_type,array_count,array1,1,arrayreal,st);
```

11.3. Program Using Within-GPR Mode

The following program initializes GPR and then initializes the 2D GMR package with fewer planes. Next the program draws a grid using GPR and displays a metafile over the grid. The program moves the metafile on the display. Next, the program adds a user-defined primitive command to the file and defines a user-defined display routine.

```
PROGRAM course4;

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';
%INCLUDE '/sys/ins/gpr.ins.pas';
%INCLUDE '/sys/ins/gmr.ins.pas';
%LIST;

CONST

    repeats = 10;
    space = 25;
    one_second = 250000;
    opcode_try_it = 1;

VAR

    bitmap_size          : gpr_$offset_t;
    init_bitmap_desc     : gpr_$bitmap_desc_t;
    st                   : status_$t;

    k                    : INTEGER;
    m                    ,
    m1                   ,
    n                    ,
    n1                  : INTEGER;

    file_id             : INTEGER;
    sid1                ,
    sid2                ,
    sid3                : gm_$segment_id_t;
    pt1                 ,
    pt2                 ,
    transl              : gm_$point16_t;
    rotate              : gm_$rotate_real2x2_t;
    rtransl             : gm_$pointreal_t;
    ptarray             : gm_$point_array16_t;
    arrayreal           : ARRAY [ 1 .. 2 ] OF real;
    pause               : time_$clock_t;


    { The actual definition of t7__try_it must be in a different
        Pascal module because ADDR( t7__try_it ) is needed. }

    PROCEDURE t7__try_it
        ( IN n_points   : INTEGER
        ; IN points     : UNIV gm_$point_array16_t
        ; IN n_param    : INTEGER
        ; IN param      : UNIV gm_$arrayreal_t
        ; OUT st        : status_$t
        ); EXTERN;
{
    VAR

        k       : INTEGER;
        width   : INTEGER;

    BEGIN

        width := TRUNC( param[ 1 ] );

        gpr_$move
            ( points[ 1 ].x - width
            , points[ 1 ].y - width
            , st
            );
        FOR k := 2 TO n_points
        DO gpr_$line
            ( points[ k ].x - width
            , points[ k ].y - width
            , st
            );

        gpr_$move
            ( points[ 1 ].x - width
            , points[ 1 ].y + width
            , st
            );
        FOR k := 2 TO n_points
        DO gpr_$line
            ( points[ k ].x - width
            , points[ k ].y + width
            , st
            );
        gpr_$move
            ( points[ 1 ].x + width
            , points[ 1 ].y - width
            , st
            );
        FOR k := 2 TO n_points
        DO gpr_$line
            ( points[ k ].x + width
            , points[ k ].y - width
            , st
            );

        gpr_$move
            ( points[ 1 ].x + width
            , points[ 1 ].y + width
            , st
            );
        FOR k := 2 TO n_points
        DO gpr_$line
            ( points[ k ].x + width
            , points[ k ].y + width
            , st
            );

        END;
}
BEGIN

    bitmap_size.x_size := 1024;
    bitmap_size.y_size := 1024;

    gpr_$init                       { Initialize GPR with 8 planes. }
        ( gpr_$borrow
        , stream_$stdout
        , bitmap_size
        , 7
        , init_bitmap_desc
        , st
        );

    gm_$init                        { Initialize 2D GMR with 3 planes. }
        ( gm_$within_gpr
        , stream_$stdout
        , pt1
        , 3
        , st
        );

    gpr_$set_draw_value             { Set GPR draw value to 9. }
        ( 9
        , st
        );

    m1 := 0;                        { Draw a grid in the GPR plane. }
    FOR m := 1 TO 8
    DO BEGIN
        m1 := m1 + 100;
        n1 := 0;
        FOR n := 1 TO 8
    DO BEGIN
        n1 := n1 + 100;
        gpr_$move
            ( m1
            , n1
            , st
            );
        gpr_$line
            ( m1+1
            , n1+1
            , st
            );
        END;
    END;

pause.low32 := 5 * one_second;
pause.high16 := 0;

time_$wait
    ( time_$relative
    , pause
    , st
    );

gpr_$set_draw_value                { You must reset the GPR draw value }
    ( 1                            { because 2D GMR will use it. }

    , st
    );

gpr_$set_plane_mask                { Change the plane mask so use of }
    ( [ 0 .. 2 ]                   { gpr_$clear does not destroy the grid. }

    , st
    );

gm_$file_create                    { Create a 2D GMR file. }
    ( 'gmfile'
    , 6
    , gm_$overwrite
    , gm_$1w
    , file_id
    , st
    );

gm_$segment_create                 { Create segment 'box.'}
    ( 'box'
    , 3
    , sid1
    , st
    );

pt1.x := 0;
pt1.y := 0;
pt2.x := 10;
pt2.y := 10;
gm_$rectangle_16                   { Add unfilled rectangle to 'box.' }
    ( pt1
    , pt2
```

```
        , FALSE
        , st
        );

    gm_$segment_close                { Close segment 'box.' }
        ( TRUE
        , st
        );

    gm_$segment_create               { Create segment 'row.' }
        ( 'row'
        , 3
        , sid2
        , st
        );

    transl.x := 0;                   { Instance segment 'box' }
    transl.y := 0;                   { repeated times. }

    FOR k := 1 TO repeats
    DO BEGIN
        transl.x := transl.x + space;
        gm_$instance_translate_2d16
            ( sid1
            , transl
            , st
            );
        END;

    gm_$segment_close                { Close segment 'row.' }
        ( TRUE
        , st
        );

    gm_$segment_create               { Create segment 'block.' }
        ( 'block'
        , 5
        , sid3
        , st
        );

    transl.x := 0;                   { Instance segment 'row' }
    transl.y := 250;                 { repeated times. }

    FOR k := 1 TO repeats
    DO BEGIN
        transl.x := transl.x + 1 ;
        transl.y := transl.y - space;
        gm_$instance_translate_2d16
            ( sid2
            , transl
            , st
            );
        END;

    gpr_$clear                       { Clear the screen. }
        ( 0
        , st
        );
```

```
rotate.xx := +   0.75;
rotate.xy :=     0.00;
rotate.yx :=     0.00;
rotate.yy := -   0.75;
rtransl.x := + 100.00;
rtransl.y := + 500.00;

gm_$display_segment_gpr_2d      { Display the rotated and translated }
    ( sid3                      { segment 'block.' }
    , rotate
    , rtransl
    , st
    );

time_$wait                      { Wait a moment. }
    ( time_$relative
    , pause
    , st
    );

gpr_$clear                      { Clear the screen. }
    ( 0
    , st
    );

rtransl.x := 200.0;
rtransl.y := 500.0;

gm_$display_segment_gpr_2d      { Display segment 'block' }
    ( sid3                      {  at a new location }

    , rotate
    , rtransl
    , st
    );

time_$wait                      { Wait a moment. }
    ( time_$relative
    , pause
    , st
    );

ptarray[ 1 ].x :=   0;          { Set parameters for the primitive }
ptarray[ 1 ].y := 300;          { command. }
ptarray[ 2 ].x := 100;
ptarray[ 2 ].y := 300;
ptarray[ 3 ].x := 100;
ptarray[ 3 ].y := 400;

arrayreal[ 1 ] := 5.0;

gm_$primitive_2d16              { Add the primitive command to }
    ( opcode_try_it             {  segment 'block.' }
    , 3
    , ptarray
    , 1
    , arrayreal
    , st
    );
```

```
gm_$primitive_display_2d        { Define the display routine for }
    ( opcode_try_it             {  the primitive command. }
    , ADDR( t7__try_it )
    , st
    );

gpr_$clear                      { Clear the screen. }
    ( 0
    , st
    );

gm_$display_segment_gpr_2d      { Display segment 'block' (includes }
    ( sid3                      {  primitive command). }
    , rotate
    , rtransl
    , st
    );

time_$wait                      { Wait a moment. }
    ( time_$relative
    , pause
    , st
    );

gm_$segment_close               { Close segment 'block.' }
    ( TRUE
    , st
    );

gm_$file_close                  { Close the file. }
    ( TRUE
    , st
    );

gm_$terminate                   { Terminate the 2D GMR package. }
    ( st
    );

END.
```

Analyzing the Program

Within-GPR mode requires you to initialize GPR before calling 2D GMR. In doing so, you retain control of the display. This means you must layout viewports and other parts of the interface. Most of the 2D GMR viewing routines are not available; you must use corresponding GPR routines instead.

When a viewing routine is called to display a segment in part of the GPR bitmap, you must specify the transformation from world coordinates to bitmap-pixel coordinates. In within-GPR mode, all the coordinates are device-dependent and use the standard GPR coordinates.

The routine GM_$DISPLAY_SEGMENT_GPR_2D is the key to the display operation. This routine is only available from within-GPR mode and is the only routine that has this limitation. This routine displays a segment that is specified as one of the parameters of the routine. The transformation is also specified in the routine.

The segment is displayed in the current GPR bitmap. This is the bitmap established when GPR and 2D GMR were initialized. At the time of display, the attributes in the current GPR attribute block are used. The clipping window and the plane mask are established by this GPR attribute block. The 2D GMR package may write to the entire Display Manager window unless you have established a clipping window and a plane mask.

Unlike the borrow, direct, and main-memory display routines, GM_$DISPLAY_SEGMENT_GPR_2D does not clear the display before drawing. If you want the display cleared, call GPR_$CLEAR.

In within_GPR mode, you can use 2D GMR attribute commands, attribute class commands, and attribute blocks. The GPR attribute block that you establish specifies the starting values of these attributes. During a 2D GMR display operation, 2D GMR attribute commands may change the GPR attribute values. However, if you use commands that change attributes during the 2D GMR display operation, the final state of the initial GPR attribute block is undefined. The 2D GMR package does not restore the starting values. Consequently, if you are going to use any 2D GMR commands that may change the GPR attribute block, you may want to make a copy of the GPR attribute block before calling GM_$DISPLAY_SEGMENT_GPR_2D.

11.4. Migration Steps from GPR to 2D GMR

Moving from GPR to GMR can involve three steps:

- Use within-GPR mode with only one segment.
- Use within-GPR mode and store graphics data in 2D GMR.
- Use the 2D GMR package fully with viewports and viewing.

Using One Segment

When you use within-GPR mode and have only one segment, the result is a simple display list processor. Information from a database is converted into graphics commands that are put into one segment. This segment is displayed, and the procedure is repeated. This first step demonstrates some of the characteristics of the 2D GMR package, but does not enhance performance.

You can display this single segment with multiple transformations. The more you redispaly this segment, the more useful it is in terms of display speed. When you are through with this segment, you can erase it and go on to the next.

You can also create a second or third segment, approaching the second step in the migration. You can use incremental redisplay in this mode when appending data: a command is put into the segment and written to the screen, and the process is repeated. You cannot erase any of the image, but you can continuously display a new segment on top of old segments. When you use within-GPR mode, the previous image is not deleted before drawing a new image. Unless you use the segment multiple times, 2D GMR in this mode does not enhance performance.

Storing Graphics Data in 2D GMR

At this stage, 2D GMR provides the advantage of rapid redisplay. You can redisplay an image multiple times at rapid display speed. In this mode you still use your exisitng GPR user interface.

A more useful approach is to store the graphics data in 2D GMR, but still use within-GPR. You can use all the 2D GMR modeling routines and create a collection of segments for redisplay as needed. You can use transformation routines to redisplay segments rapidly. In addition, the instance commands are all available. This allows rapid redisplay speed without rewriting an existing GPR user interface.

Using the 2D GMR package

The final step is to use the 2D GMR package completely.

Chapter 12
Output

The chapter describes the routines and external file format used in generating hard-copy output of graphics data.

12.1. Printing

Functions:

```
GM_$PRINT_FILE
GM_$PRINT_FILE_PART
```

The print file routines enable you to generate files for printing on a hard-copy device. These routines copy part or all of a 2D GMR to a bitmap, then store that bitmap in a GMF (graphics map file) or in a 2D GMR vector command file.

You can specify the size in pixels of the bitmap you want created. For a GMF, you may also specify the scale at which the output device (e.g., printer) is to attempt to print the bitmap. You may print GMFs using the Shell command PRF with the -PLOT option. You may print 2D GMR vector command files by writing a driver for a particular device, using the format information provided in the next section.

The format for vector command files is described in this chapter. For a description of GMF routines, see the *DOMAIN System Call Reference*, Volume 1.

GM_$PRINT_FILE converts the current metafile to the specified file for subsequent printing on a hard-copy device.

GM_$PRINT_FILE_PART converts part of the current metafile to the specified file for subsequent printing on a hard-copy device. The part converted is within the physical bounds you specify, in terms of segment coordinates of the primary segment of the metafile.

12.2. External File Format

You can create 2D GMR vector command files using GM_$PRINT_... routines. The format of these files is described in this section. Using this format, you can write a device driver for printing vector command files on a hard-copy output device of your choice, for example a pen plotter.

The file created looks like a sequence of 2D GMR commands as stored in a 2D GMR segment. The command formats and op codes are described below. All output coordinate data is (x,y) pairs of 16-bit integers.

The file or segment is flattened into a single list of commands. All coordinates are transformed to display coordinates in accordance with the size parameter of the GM_$PRINT_... routine that you used to create the vector command file. In the GM_$OUT1 file, the origin of coordinates is the top left, not the bottom left as in the metafile. The GM_$OUT1 file is scaled to the size parameter, using the standard 95% rule that one dimension fills 95% of the size block, and the other dimension does not overflow the block.

Individual commands are not clipped to the display size; your display driver must perform this function itself.

The following occurs in the translation to the vector command file:

- All instance commands are resolved.
- Stroke text commands are decomposed into individual primitive commands.
- Attribute class commands are decomposed into individual attribute commands.
- Tag commands are discarded.

Nonvector commands, such as pixel text commands, are passed through untouched; your device driver may use them or throw them away.

Format Details

The vector command file begins with a 32-byte file header that contains the length of the command portion of the file (that is, everything but this header) and the size you specified when creating the file.

```
HEADER

    bytes 0-3  : length of command portion of the file,
                 in bytes (4-byte integer)

    bytes 4-5  : the x size specified when creating the file
                 (2-byte integer)

    bytes 6-7  : the y size specified when creating the file
                 (2-byte integer)

    bytes 8-31 : unused
```

The command portion of the file starts with byte 32. It contains command op codes and data, interspersed. All op codes are 16-bit integers; all coordinate data is 16-bit integers. All op codes are aligned on 16-bit word boundaries.

The formats of the commands (their op codes and their data) are as follows:

```
COMMANDS

END OF FILE

    bytes 0-1 : 16#0000 (op code for end-of-file)
```

```
   TOTAL LENGTH OF COMMAND = 2 bytes

POLYLINE

   bytes 0-1 : 16#0020 (op code for polyline)

   bytes 2-3 : number of points (2-byte integer)

   bytes 4-5 : first x-coordinate (2-byte integer)

   bytes 6-7 : first y-coordinate (2-byte integer)

   bytes 8-11,... : additional (x,y) pairs, each a pair
                    of 2-byte integers

   TOTAL LENGTH OF COMMAND = 4 * (n_points + 1) bytes


CLOSED POLYLINE

   bytes 0-1 : 16#0021 (op code for closed polyline)

   bytes 2-... : same as POLYLINE

   TOTAL LENGTH OF COMMAND = 4 * (n_points + 1) bytes


FILLED POLYLINE

   bytes 0-1 : 16#0022 (op code for filled polyline)

   bytes 2-... : same as POLYLINE

   TOTAL LENGTH OF COMMAND = 4 * (n_points + 1) bytes


RECTANGLE

   bytes 0-1 : 16#0030 (op code for rectangle)

   bytes 2-3 : first x-coordinate (2-byte integer)

   bytes 4-5 : first y-coordinate (2-byte integer)

   bytes 6-7 : second x-coordinate (2-byte integer)

   bytes 8-9 : second y-coordinate (2-byte integer)

   TOTAL LENGTH OF COMMAND = 10 bytes


FILLED RECTANGLE

   bytes 0-1 : 16#0031 (op code for filled rectangle)
   bytes 2-9 : same as RECTANGLE

   TOTAL LENGTH OF COMMAND = 10 bytes


CIRCLE

   bytes 0-1 : 16#0040 (op code for circle)

   bytes 2-3 : center x-coordinate (2-byte integer)

   bytes 4-5 : center y-coordinate (2-byte integer)

   bytes 6-7 : radius (2-byte integer)

   TOTAL LENGTH OF COMMAND = 8 bytes


FILLED CIRCLE

   bytes 0-1 : 16#0041 (op code for filled circle)

   bytes 2-7 : same as CIRCLE

   TOTAL LENGTH OF COMMAND = 8 bytes


CURVE

   bytes 0-1 : 16#0050 (op code for curve)

   bytes 2-3 : curve type (2-byte integer;
                           0 = parametric cubic spline,
                           1 = 3-pt arc)

   bytes 4-5 : number of points (2-byte integer)

   bytes 6-7 : number of parameters (2-byte integer)

   bytes 8-9 : first x-coordinate (2-byte integer)

   bytes 10-11 : first y-coordinate (2-byte integer)

   bytes 12-15,... : additional (x,y) pairs, each a pair
                     of 2-byte integers

   bytes (4*n_points + 8)-(4*n_points + 11) : first parameter
                                              (real)

   bytes (4*n_points + 12)-... : additional parameters,
                                 each a real value

   TOTAL LENGTH OF COMMAND =
         8 + 4 * (n_points + n_parameters) bytes
```

USER-DEFINED PRIMITIVE

```
bytes 0-1 : 16#0060 (op code for user-defined primitive)

bytes 2-3 : user-defined primitive type
            (values assigned by user)

bytes 4-... : same as CURVE

TOTAL LENGTH OF COMMAND =
      8 + 4 * (n_points + n_parameters) bytes
```

PIXEL TEXT

```
bytes 0-1 : 16#0070 (op code for pixel text)

bytes 2-3 : text location x-coordinate (2-byte integer)

bytes 4-5 : text location y-coordinate (2-byte integer)

bytes 6-9 : text rotation in degrees (real)

bytes 10-11 : number of characters (2-byte integer)

byte 12 : first character

bytes 13-... : additional characters

If the number of characters is odd, an unused byte
is appended to keep subsequent commands aligned on
16-bit word boundaries.

TOTAL LENGTH OF COMMAND = 12 + n_characters bytes,
      plus 1 if n_characters is odd
```

DRAW VALUE

```
bytes 0-1 : 16#0080 (op code for draw value)

bytes 2-5 : draw value (4-byte integer)

TOTAL LENGTH OF COMMAND = 6 bytes
```

DRAW STYLE

```
bytes 0-1 : 16#0081 (op code for draw style)

bytes 2-3 : draw style (2-byte integer;
                        0 = solid,
                        1 = dotted,
                        2 = patterned)

            for solid: ignored
```

```
              for dotted: length of solid and blank portions
                          of line
              for patterned:
bytes 4-5  :  replication factor; the number of times
              each bit in the pattern is to be repeated
              (2-byte integer)

bytes 6-7  :  number of bits in the pattern
              (2-byte integer)

bytes 8-15 :  the pattern of bits; only the first n_bits
              of the bits are significant (an array of
              4 2-byte integers)

TOTAL LENGTH OF COMMAND = 16 bytes
```

DRAW RASTER OP

```
bytes 0-1 : 16#0082 (op code for draw raster op)

bytes 2-3 : draw raster op (2-byte integer)

TOTAL LENGTH OF COMMAND = 4 bytes
```

PLANE MASK

```
bytes 0-1 : 16#0083 (op code for plane mask)

bytes 2-3 : plane mask; see description for FORTRAN users
            in the description of GM_$PLANE_MASK in
            Chapter 16 (2-byte integer)

TOTAL LENGTH OF COMMAND = 4 bytes
```

FILL VALUE

```
bytes 0-1 : 16#0090 (op code for fill value)

bytes 2-5 : fill value (4-byte integer)

TOTAL LENGTH OF COMMAND = 6 bytes
```

FILL BACKGROUND VALUE

```
bytes 0-1 : 16#0091 (op code for fill background value)

bytes 2-5 : fill background value (4-byte integer)

TOTAL LENGTH OF COMMAND = 6 bytes
```

FILL PATTERN

```
bytes 0-1 : 16#0092 (op code for fill pattern)

{ FORMAT : OP CODE    scale size pattern }
            If fill pattern is solid (that is, no fill pattern),
            these two bytes are zero, and the other bytes are
            ignored.

bytes 2-3 : scale; the number of times each bit in the
            pattern is to be repeated (2-byte integer)
            (Currently, always 1)

bytes 4-5 : x size; the number of bits in each row of
            the pattern (2-byte integer)
            (Currently, always 32)

bytes 6-7 : y size; the number of rows in the pattern
            (2-byte integer)
            (Currently, always 32)

bytes 8-135 : the pattern of bits (32 4-byte integers;
              each represents one row of the pattern)

TOTAL LENGTH OF COMMAND = 136 bytes
```

TEXT VALUE

```
bytes 0-1 : 16#00A0 (op code for text value)

bytes 2-5 : text value (4-byte integer)

TOTAL LENGTH OF COMMAND = 6 bytes
```

TEXT BACKGROUND VALUE

```
bytes 0-1 : 16#00A1 (op code for text background value)

bytes 2-5 : text background value (4-byte integer)

TOTAL LENGTH OF COMMAND = 6 bytes
```

TEXT SIZE

```
bytes 0-1 : 16#00A2 (op code for text size)

bytes 2-3 : text size in display pixels (2-byte integer)

TOTAL LENGTH OF COMMAND = 4 bytes
```

```
FONT FAMILY

   bytes 0-1 : 16#00A3 (op code for font family)

   bytes 2-3 : font family id number (2-byte integer)

   TOTAL LENGTH OF COMMAND = 4 bytes
```

Chapter 13
Attribute Classes and Blocks

This chapter describes the use of attribute classes and blocks and explains how to tie attribute blocks to attribute classes for the entire display and for individual viewports. Programming examples illustrate the use of attribute routines.

13.1. Terms Used with Attributes

Chapter 5 describes how to insert attribute commands into the metafile to affect the appearance of subsequent primitive commands when they are displayed. You can use these attribute commands to change characteristics such as text size, line style, and background. The display of these primitive commands is in accordance with the values you assign to the attributes.

The 2D GMR package also has a more powerful mechanism for handling attributes: attribute classes. The following is an outline of the procedure for using attribute classes. The references direct you to sections describing the parts of the procedure.

- Create attribute classes (aclasses) (Section 13.2).
- Assign attributes to attribute blocks (ablocks) (Sections 13.3 through 13.5).
- Assign attribute blocks to attribute classes for display (Section 13.9).

A program at the end of this chapter illustrates the use of attribute classes and blocks. Before reading in detail, skim through this entire chapter for a sense of the relationship of the steps in the process. Then go back and read in detail.

Terms Defined

- An attribute class command in a metafile indicates that you want the attribute values changed to values you defined elsewhere. You define the attribute values to which the command corresponds either elsewhere in the file, or when the file is displayed.
- An attribute block is a collection of attribute values. You can use an attribute block to assign attribute values to attribute classes when the file is displayed.

13.2. Using Attribute Classes

Function:

GM_$ACLASS

Attribute classes allow you to use attributes by changing between collections of attributes, rather than changing each attribute each time. This is useful, for example, when you want to display different layers of a printed circuit board using different attributes. You can assign each layer a distinct attribute class number. You can then include in the file numerous commands to switch to a new attribute class.

Attribute class commands in a segment are signals to the 2D GMR package to switch among collections of attributes. These collections are read from attribute blocks as the segment is being displayed. You use attribute block routines to define the attributes associated with each collection when the file is displayed (see Sections 13.3 through 13.6).

The GM_$ACLASS routine inserts into the metafile a command indicating that the attributes currently associated with that attribute class are to be used when displaying subsequent primitive commands. For example:

```
    gm_$aclass(5,status);
    gm_$circle_16(center,10,false,status);
    gm_$aclass(7,status);
    gm_$circle_16(center,20,false,status);
```

The above sequence of routines inserts into the metafile four commands (use aclass 5, draw a circle of radius 10, use aclass 7, draw a circle of radius 20). When this sequence of commands is displayed, the small circle is displayed using the attributes associated with aclass 5, and the large circle and subsequent commands are displayed using the attributes associated with aclass 7.

At the start of a file, the default attribute class number is 1. This default is used until another class is designated using the command GM_$ACLASS.

An attribute class is a means of referring to a collection of attribute values. The particular attribute values are defined elsewhere.

The procedure for assigning attributes to an attribute class is as follows. You may define attribute blocks and then use viewing routines to associate attribute blocks with attribute classes. Your input to the viewing routines is the identification of the attribute class and the attribute block to associate with the class. This association of attribute class and attribute block may be for the display as a whole or for individual viewports (see Section 13.9).

The above procedure allows you to attach attribute classes to different sets of attributes. The choice of attributes depends on the type of node you are using.

This procedure also allows the user of the application to do the following:

- Interactively modify attributes used to display the file without affecting the contents of the file.

- Assign different attributes to an attribute class in different viewports.

If you do not assign attribute values to a particular attribute class, the default attribute values are used (see Table 3-1).

13.3. Creating Attribute Blocks

Function:

GM_$ABLOCK_CREATE

To associate collections of attributes with attribute class numbers, you must first build an attribute block.

The attribute values in an attribute block define a set of characteristics that affect the appearance of the picture. An attribute block is a data structure that holds a collection of attribute values in a form allowing you to modify or inquire about individual attributes. These attributes include draw, fill, and text values; raster op codes; and the plane mask.

Attribute block 1 contains the default collection of attribute values to which the package is initialized (see Table 3-1). You can use attribute block 1 as a starting point for creating new attribute blocks. However, you may not modify this default attribute block 1.

To create a collection of attributes, you first define an attribute block using the routine GM_$ABLOCK_CREATE. GM_$ABLOCK_CREATE creates an attribute block identical to a specified existing attribute block, such as the default (ablock 1), and assigns a new ablock identification number to it. You then change attribute values in it using GM_$ABLOCK_SET... routines. For example:

```
gm_$ablock_create(1,ablockid,status);
gm_$ablock_set_draw_value_(ablockid,2,status);
```

These routines create a new ablock, which the 2D GMR package assigns the identification number "ablockid." This new ablock contains all of the default attribute values except for the draw value, which has been changed to 2 by the second routine above.

You can then use the routine GM_$ABLOCK_ASSIGN_DISPLAY (defined in Section 13.9) to associate this new ablock with a particular aclass. For example, to associate this ablock "ablockid" with aclass 5, use the following:

```
gm_$ablock_assign_display(5,ablockid,status);
```

When commands are subsequently displayed, a command to "use aclass 5" causes this collection of attribute values to be applied when subsequent primitive commands are displayed.

You can use the routine GM_$ABLOCK_ASSIGN_VIEWPORT (described in Section 13.9) to associate a single aclass with different ablocks in different viewports.

13.4. Modifying Attribute Blocks

Functions:

```
GM_$ABLOCK_SET_DRAW_VALUE
GM_$ABLOCK_SET_DRAW_STYLE
GM_$ABLOCK_SET_DRAW_RASTER_OP
GM_$ABLOCK_SET_FILL_VALUE
GM_$ABLOCK_SET_FILL_PATTERN
GM_$ABLOCK_SET_PLANE_MASK
GM_$ABLOCK_SET_TEXT_VALUE
GM_$ABLOCK_SET_TEXT_BACKGROUND_VALUE
GM_$ABLOCK_SET_TEXT_SIZE
GM_$ABLOCK_SET_FONT_FAMILY
```

To change attributes in an attribute block, you use a different routine for each attribute to be changed, identifying the attribute block that you want to change.

For each attribute, there is a "no-change" value (usually -1). When an attribute class refers to an attribute block with this value, the previous attribute value remains unchanged. This allows you to define attribute blocks that keep certain values constant while you change others. In this way, you can preserve existing attributes across changes in the attribute class without having to set the attributes explicitly each time.

A complete set of these "no-change" attribute values is stored in attribute block 0. Thus, assigning attribute block 0 to an attribute class would be a null operation; that is, it would change no attribute values. As with attribute block 1, you may copy attribute block 0, but not change it. For a list of these "no-change" attribute values, see Table 13-1.

13.5. Reading Attribute Blocks

Functions:

```
GM_$ABLOCK_INQ_DRAW_VALUE
GM_$ABLOCK_INQ_DRAW_STYLE
GM_$ABLOCK_INQ_DRAW_RASTER_OP
GM_$ABLOCK_INQ_FILL_VALUE
GM_$ABLOCK_INQ_FILL_PATTERN
GM_$ABLOCK_INQ_PLANE_MASK
GM_$ABLOCK_INQ_TEXT_VALUE
GM_$ABLOCK_INQ_TEXT_BACKGROUND_VALUE
GM_$ABLOCK_INQ_TEXT_SIZE
GM_$ABLOCK_INQ_FONT_FAMILY
```

The routines listed above return the current values of an individual attribute in the specified attribute block.

The default attribute values are shown in Table 3-1. The "no-change" attribute values are shown in Table 13-1.

13.6. Copying Attribute Blocks

Functions:

```
GM_$ABLOCK_COPY
```

Once you have assigned attributes to the attribute block, you may want to copy these attributes to an existing attribute block. To do this, use GM_$ABLOCK_COPY. To establish a new attribute block identical to it, use GM_$ABLOCK_CREATE as described previously.

13.7. Instancing and Attributes

Chapter 5 explains that commands that change individual attribute values affect all subsequent commands in that segment. The same applies to a GM_$ACLASS command: the command

Table 13-1. "No-Change" Attribute Values

ATTRIBUTE	VALUE
Draw Style	GM_$SAME_DRAW_STYLE
Draw Value	-1
Fill Value	-1
Fill Background Value	-3
Fill Pattern	(scale = -1)
Text Value	-1
Text Background Value	-3
Text Size	-1.0
Font Family ID Number	-1
Plane Mask	(change = false)
Draw Raster Op	-1

changes the attribute values applied to all subsequent commands in that segment. This includes any other segments referenced using instance commands. When display of the segment containing the GM_$ACLASS command is completed, the previous attribute class is restored before the display of commands in the instancing segment continues.

In other words, like attribute commands and segment transformations, attribute values are affected forward and downward in the hierarchy of segments and commands, but never upward. This allows different instancing segments to apply different attributes to a particular instanced segment.

13.8. Mixing Attribute Commands and Attribute Classes

When an attribute command is encountered, it overrides all data for that attribute in all attribute classes. Subsequent changes of attribute class do not affect that attribute. For example:

```
gm_$draw_value (4, status);
gm_$aclass (any_class, status);
```

After the two commands above, the draw value will be 4 regardless of the draw value in the attribute block that you have assigned to the attribute class any_class. The draw value 4 is effectively copied into the current definition of all attribute classes. The draw value remains in effect until the end of the current segment is reached.

13.9. Attributes and Viewing Operations

To change attributes, you may insert individual attribute commands into a file (see Chapter 5). Alternatively, you can insert attribute classes into a file, as described in this chapter. You can define attribute blocks and apply them to attribute classes, either for the GM bitmap, or in individual viewports. You can also apply a particular attribute block to one or more viewports (see Section 13.11 for a program example).

13.9.1. Tying Ablocks to Aclasses for the Entire GM Bitmap

```
Functions:

GM_$ABLOCK_ASSIGN_DISPLAY
GM_$ABLOCK_INQ_ASSIGN_DISPLAY
```

You can change all attributes at once on the GM bitmap as a whole. To do this, use GM_$ACLASS to insert commands into the file to change the attribute class to be used. In this routine, you specify an attribute class identification. The command is not effective until you associate it with an attribute block.

You can assign attribute blocks to attribute class numbers for display purposes using GM_$ABLOCK_ASSIGN_DISPLAY. Your input to the display routine is this same aclass identification along with the identification of the attribute block you want to use. When commands are being displayed, the following occurs: When an "ACLASS" command is encountered, the attributes of the attribute block assigned to this class are subsequently used.

You may develop a program that creates attribute blocks and assigns them to attribute classes. You can use such a program to display pictures that you have already created. At the time you view the file, you use GM_$ABLOCK_ASSIGN_DISPLAY to associate the attribute class you identified with the attribute block you want used.

GM_$ABLOCK_ASSIGN_DISPLAY assigns an attribute block (by number) to an attribute class, for the entire display.

GM_$ABLOCK_INQ_ASSIGN_DISPLAY returns the current attribute block number assigned to a particular attribute class for the display.

13.9.2. Tying Ablocks to Aclasses for Individual Viewports

```
Functions:

GM_$ABLOCK_ASSIGN_VIEWPORT
GM_$ABLOCK_INQ_ASSIGN_VIEWPORT
```

You can change all attributes at once for individual viewports of the display. The attributes in each viewport may be different. To change the attributes used in a viewport, do the following:

- Use GM_$ACLASS to insert commands into the file to specify the attribute class you want used. In this routine, you specify an attribute class identification. The command is not effective until you associate it with an attribute block.

- You can view the file using GM_$ABLOCK_ASSIGN_VIEWPORT. Your input to the display routine is this same attribute class identification, along with the identification of the attribute block and the viewport.

Note that if an attribute block is specified for a viewport, it overrides the specification of an attribute block for the GM bitmap.

GM_$ABLOCK_ASSIGN_VIEWPORT assigns an attribute block (by number) to an attribute class, for the specified viewport.

GM_$ABLOCK_INQ_ASSIGN_VIEWPORT returns the current attribute block number assigned to a particular attribute class, for the specified viewport.

13.9.3. Summary of Procedures

Use the following procedures to establish attribute blocks and assign them to the GM bitmap.

- Use GM_$ABLOCK_CREATE to create an attribute block equivalent to the source block you identify. The routine returns the attribute block identification number.

- Change the attribute block with the calls GM_$ABLOCK_SET.... In these calls, you specify the value of the attribute and identify the attribute block to which it belongs.

- Use GM_$ABLOCK_ASSIGN_DISPLAY to assign the attribute block to a class for the GM bitmap. This assignment is used for all viewports until you assign an attribute block to a class for a particular viewport using GM_$ABLOCK_ASSIGN_VIEWPORT.

- You may subsequently change attribute values in the assigned attribute blocks. When you next display the picture, the result is the following: the new attribute values assigned to this attribute block are used whenever an attribute class command associated with the attribute block is encountered.

13.10. Summary

The 2D GMR package provides three techniques for modifying attributes. The first technique assigns individual attribute values within a file.

- Change one attribute at a time within the file. To do this, put commands into the file to change individual attributes.

The next two techniques associate attributes with attribute classes only when the file is displayed. Neither the attribute values nor the class assignment is stored in the file.

- Change all attributes at once on the display as a whole. To do this, use GM_$ACLASS to put commands into the file to specify the attribute class you want used. Then while viewing the file, change the collection of attributes assigned to attribute classes. To make this change, use GM_$ABLOCK_ASSIGN_DISPLAY, as described in Section 13.9.

- Change all attributes at once for individual viewports of the display. The attributes in each viewport may be different. To do this, use GM_$ACLASS to put commands into a file to specify the attribute class. Then while viewing the file, change the collection of attributes assigned to attribute classes for individual viewports. To make this change, use GM_$ABLOCK_ASSIGN_VIEWPORT, as described in Section 13.9.

13.11. Program with Attribute Classes and Blocks

The following program creates a hierarchy of segments including instance commands. It displays the file in three viewports; adds attribute class commands to the file; assigns attribute blocks to attribute classes; displays the segments; closes the file; and terminates the package.

```
PROGRAM course5;

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';
%INCLUDE '/sys/ins/gmr.ins.pas';
%LIST;

CONST

    aclassid1    = 1;        { Default aclass }
    aclassid2    = 2;
    vpid1        = 1;        { Initial viewport }
    one_second   = 250000;

VAR

    bitmap_size       : gm_$point16_t;
    status            : status_$t;

    b                 : gm_$boundsreal_t;
    vpid2             ,
    vpid3             : INTEGER;

    file_id           : INTEGER;
    sid1              ,
    sid2              ,
    sid3              : gm_$segment_id_t;
    ablockid1         ,
    ablockid2         ,
    ablockid3         : INTEGER;
    ablockid4         ,
    ablockid5         ,
    ablockid6         : INTEGER;

    pt1               ,
    pt2               : gm_$point16_t;

    pattern           : gm_$draw_pattern_t;
    pause             : time_$clock_t;
```

```
BEGIN

    bitmap_size.x := 1024;
    bitmap_size.y := 1024;

    gm_$init                        { Initialize the 2D GMR package. }
        ( gm_$borrow
        , stream_$stdout
        , bitmap_size
        , 8
        , status
        );

    b.xmin := 0.00;
    b.ymin := 0.00;
    b.xmax := 0.49;
    b.ymax := 0.49;
    gm_$viewport_set_bounds         { Create viewport 1. }
        ( b
        , status
        );

    b.xmin := 0.51;
    b.ymin := 0.00;
    b.xmax := 1.00;
    b.ymax := 0.49;
    gm_$viewport_create             { Create viewport 2. }
        ( b
        , vpid2
        , status
        );

    b.xmin := 0.00;
    b.ymin := 0.51;
    b.xmax := 1.00;
    b.ymax := 1.00;
    gm_$viewport_create             { Create viewport 3. }
        ( b
        , vpid3
        , status
        );

    gm_$file_create                 { Create and name a metafile. }
        ( 'gmfile'
        , 6
        , gm_$overwrite
        , gm_$1w
        , file_id
        , status
        );

    gm_$segment_create              { Create segment 'bottom.' }
        ( 'bottom'
        , 6
        , sid1
        , status
        );

    pt1.x := 0;
```

```
pt1.y := 30;
pt2.x := 10;
pt2.y := 40;
gm_$rectangle_16            { Add a rectangle to segment 'bottom.' }
    ( pt1
    , pt2
    , FALSE
    , status
    );

gm_$draw_style              { Change the draw style to solid. }
    ( gm_$solid
    , 0
    , pattern
    , 0
    , status
    );

pt1.x := 20;
pt2.x := 30;
gm_$rectangle_16            { Add a rectangle to segment 'bottom.' }
    ( pt1
    , pt2
    , FALSE
    , status
    );

gm_$segment_close           { Close segment 'bottom.' }
    ( TRUE
    , status
    );

gm_$segment_create          { Create segment 'top.'. }
    ( 'top'
    , 3
    , sid2
    , status
    );

pt1.x := 0;
pt1.y := 0;
pt2.x := 10;
pt2.y := 10;
gm_$rectangle_16            { Add a rectangle to segment 'top.' }
    ( pt1
    , pt2
    , FALSE
    , status
    );

gm_$instance_translate_2d16 { Instance segment 'bottom' into segment 'top.' }
    ( sid1
    , pt1
    , status
    );

pt1.x := 20;
pt2.x := 30;
gm_$rectangle_16            { Add a rectangle to segment 'top.' }
```

```
    ( pt1
    , pt2
    , FALSE
    , status
    );
gm_$segment_close           { Close segment 'top.' }
    ( TRUE
    , status
    );

gm_$display_file            { Display the file in viewport 3. }
    ( status
    );

gm_$viewport_select         { Select viewport 2. }
    ( vpid2
    , status
    );

gm_$display_file            { Display the file in viewport 2. }
    ( status
    );

gm_$viewport_select         { Select viewport 1. }
    ( vpid1
    , status
    );

gm_$display_file            { Display file in viewport 1}
    ( status
    );

pause.low32 := 5 * one_second;
pause.high16 := 0;
time_$wait
    ( time_$relative
    , pause
    , status
    );

gm_$ablock_create           { Create ablockid1. }
    ( 1
    , ablockid1
    , status
    );

gm_$ablock_set_draw_style   { Give ablockid1 the dotted line style }
    ( ablockid1             {   with repetition factor = 5. }
    , gm_$dotted
    , 5
    , pattern
    , 0
    , status
    );

gm_$ablock_assign_viewport  { Assign ablockid1 to aclassid1 in viewport 1. }
    ( aclassid1
    , vpid1
    , ablockid1
    , status
    );

gm_$ablock_create            { Create ablockid2. }
    ( 1
    , ablockid2
    , status
    );

gm_$ablock_set_draw_style    { Give ablockid2 the dotted line style }
    ( ablockid2              {   with repetition factor = 10. }
    , gm_$dotted
    , 10
    , pattern
    , 0
    , status
    );

gm_$ablock_assign_viewport   { Assign ablockid1 to aclassid2 in viewport 2. }
    ( aclassid1
    , vpid2
    , ablockid2
    , status
    );

gm_$ablock_create            { Create ablockid3. }
    ( 1
    , ablockid3
    , status
    );

gm_$ablock_set_draw_style    { Give ablockid3 the dotted line style. }
    ( ablockid3              {   with repetition factor = 20 }
    , gm_$dotted
    , 20
    , pattern
    , 0
    , status
    );

gm_$ablock_assign_viewport   { Assign ablockid3 to aclassid1 in viewport 3. }
    ( aclassid1
    , vpid3
    , ablockid3
    , status
    );

gm_$display_refresh          { Refresh display to see the effects of }
    ( status                 {   the attribute blocks. }
    );

time_$wait
    ( time_$relative
    , pause
    , status
    );

gm_$segment_create           { Create segment 'new.'. }
    ( 'new'
    , 3
    , sid3
    , status
    );

pt1.x := 0;
pt1.y := 0;
pt2.x := 10;
pt2.y := 10;
gm_$rectangle_16            { Add a rectangle to segment 'new.' }
    ( pt1
    , pt2
    , FALSE
    , status
    );

gm_$aclass                  { Add an aclass command to segment 'new.' }
    ( aclassid2
    , status
    );

pt1.x := 20;
pt2.x := 30;
gm_$rectangle_16            { Add a rectangle to segment 'new.' }
    ( pt1
    , pt2
    , FALSE
    , status
    );

gm_$segment_close           { Close segment 'new.' }
    ( TRUE
    , status
    );

gm_$ablock_create           { Create ablockid4. }
    ( 1
    , ablockid4
    , status
    );

gm_$ablock_set_draw_style   { Give ablockid4 the dotted line style }
    ( ablockid4             {   with repetition factor = 30. }
    , gm_$dotted
    , 30
    , pattern
    , 0
    , status
    );

gm_$ablock_assign_viewport  { Assign ablockid4 to aclassid2 in viewport 1. }
    ( aclassid2
    , vpid1
    , ablockid4
    , status
    );

gm_$ablock_create           { Create ablockid5. }
    ( 1
        , ablockid5
        , status
        );

    gm_$ablock_set_draw_style   { Give ablockid5 the dotted line style }
        ( ablockid5             {   with repetition factor = 40. }
        , gm_$dotted
        , 40
        , pattern
        , 0
        , status
        );

    gm_$ablock_assign_viewport  { Assign ablockid5 to aclassid2 in viewport 2. }
        ( aclassid2
        , vpid2
        , ablockid5
        , status
        );

    gm_$ablock_create           { Create ablockid6. }
        ( 1
        , ablockid6
        , status
        );

    gm_$ablock_set_draw_style   { Give ablockid6 the dotted line style }
        ( ablockid6             {   with repetition factor = 50. }
        , gm_$dotted
        , 50
        , pattern
        , 0
        , status
        );

    gm_$ablock_assign_viewport  { Assign ablockid6 to aclassid2 in viewport 3. }
        ( aclassid2
        , vpid3
        , ablockid6
        , status
        );

    gm_$display_segment         { Display  segment 'new' in viewport 1. }
        ( sid3
        , status
        );

    gm_$viewport_select         { Select viewport 2. }
        ( vpid2
        , status
        );

    gm_$display_segment         { Display segment 'new' in viewport 2. }
        ( sid3
        , status
        );

    gm_$viewport_select         { Select viewport 3. }
        ( vpid3
    , status
    );

gm_$display_segment          { Display segment 'new' in viewport 3. }
    ( sid3
    , status
    );

time_$wait
    ( time_$relative
    , pause
    , status
    );

gm_$file_close               { Close the file. }
    ( TRUE
    , status
    );

gm_$terminate                { Terminate the 2D GMR package. }
    ( status
    );

END.
```

Chapter 14
Advanced Display Techniques

This chapter describes advanced display techniques including using color as well as viewport border and background. Programming examples illustrate these techniques.

14.1. Using the Color Map

Graphics programs use a color map to specify color and intensity (gray-scale) values. A program can redefine the color map to assign colors to pixel values. (On a monochromatic node, you can only switch the definitions of black and white). To assign different colors to lines or other graphic entities, a program must draw them using different pixel values and then assign the appropriate colors to these pixel values. You can assign a pixel value (color map index) to the draw value attribute, the text value attribute, the text background value attribute, and the fill value attribute.

In within-GPR mode, you must call GPR routines to assign values to, or read values from, the color map.

14.1.1. The Color Map: A Set of Color Values

The color map is a display feature, not an attribute. This means that you cannot specify a color map in a metafile. However, the color map can be stored as tag data and read by an application program. You can specify only one color map for the display.

A color map is a set of color values, each representing a color and intensity. A color value is an encoding of a particular visible color/intensity, based on the RGB (red/green/blue) color model. The RGB color model defines red, green, and blue as primary colors. All other colors are combinations of these primaries, including the three secondary colors (cyan, magenta, and yellow).

Each color value consists of three component values, each a real number in the range 0.0 to 1.0. The first real number is the value for the red component of the color; the second, the green component; and the third, the blue component. A value of 0.0 specifies the absence of the primary color, and a value of 1.0 specifies full intensity of that primary color.

On a color display, the red, green, and blue component values are displayed as accurately as possible, depending on the possible color values available on the node. For a detailed description of color and display configurations, see *Programming with DOMAIN Graphics Primitives*.

On a monochrome display, you may either assign black (0.0, 0.0, 0.0) to 0 and white (1.0, 1.0, 1.0) to 1, or vice versa.

If all three component values are equal, the color value is a shade of gray, as Table 14-1 shows.

Table 14-1. Example of Gray-Scale Color Values and Visible Intensities

Color Value			Visible Color/Intensity
R value	G value	B value	
1.0	1.0	1.0	white
0.75	0.75	0.75	light gray
0.5	0.5	0.5	medium gray
0.25	0.25	0.25	dark gray
0.0	0.0	0.0	black

A color map consists of a set of color map entries; each entry is a color value associated with an index. Though the association between color values and visible colors/intensities cannot be changed, a program can establish and change the association between indexes and color values by changing the entries in the color map. In this way, a program can select the set of colors/intensities to constitute a color map for a particular application, and associate them with particular indexes.

For an eight-plane color display, the color map has 256 entries, with index values 0-255. For a four-plane color display, the color map has 16 entries, with index values 0-15. For all displays, all entries are set to default values at the initialization of the 2D GMR package (see Figure 14-1).

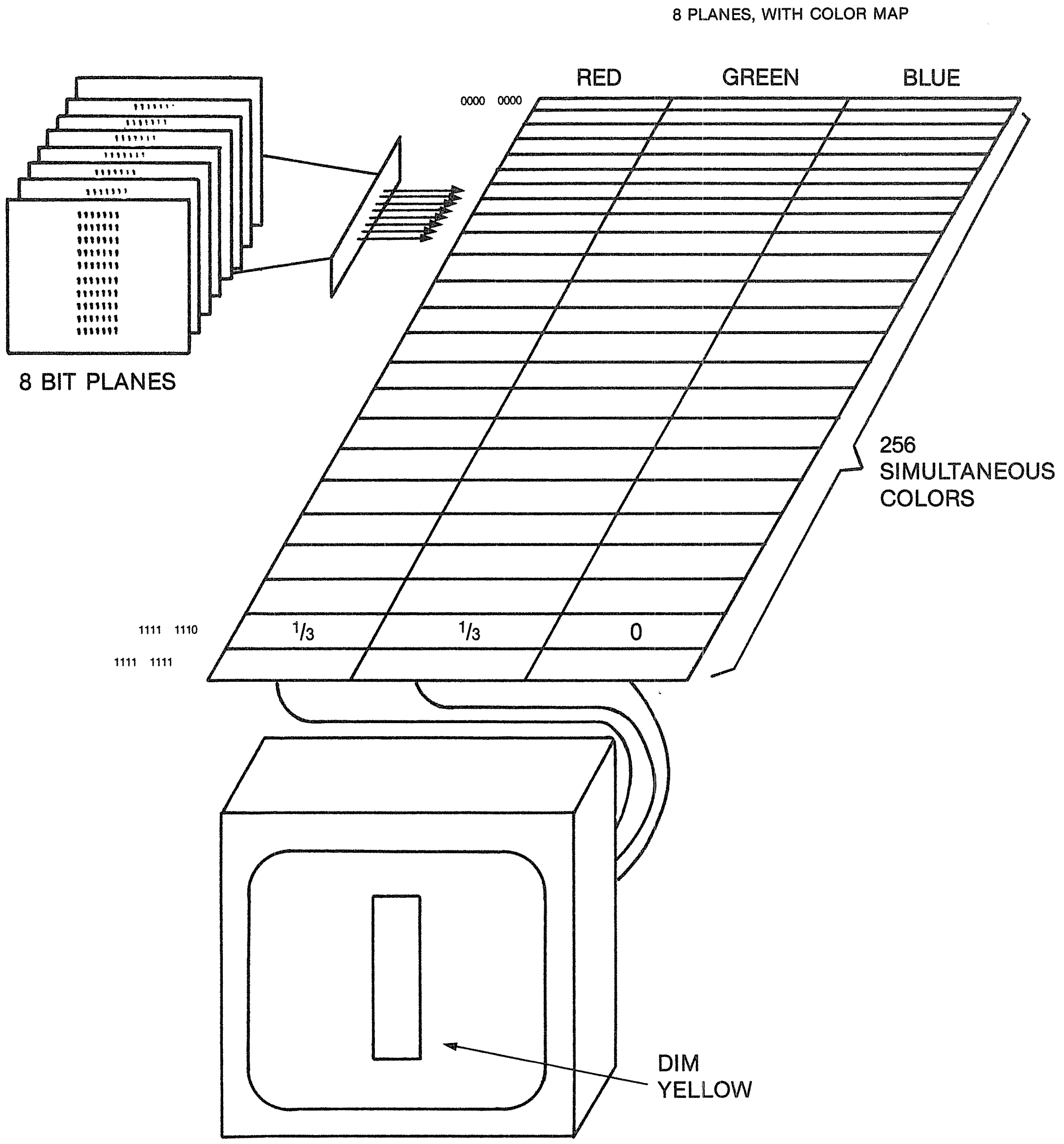


Figure 14-1. The Pixel Value Used as an Index into the Color Map

Table 14-2 shows the default color map.

Table 14-2. Default Color Map

Monochromatic displays have only the first two color map entries. Four-plane color displays have only the first sixteen color map entries.

Color Table Index	Color Value	Resultant Visible Color/Intensity
	R G B	
0	0.0 0.0 0.0	black
1	1.0 1.0 1.0	white
2	0.0 1.0 0.0	green
3	0.0 0.0 1.0	blue
4	0.0 1.0 1.0	cyan
5	1.0 1.0 0.0	yellow
6	1.0 0.0 1.0	magenta
7	1.0 1.0 1.0	white
8-15	Contain colors used by the Display Manager to display windows.	
16-255	0.0 0.0 0.0	black

In direct mode, a program cannot modify color map entries 0 and 7-15. Thus, the color map entries that may be changed are the following (Table 14-3):

Table 14-3. Color Map Entries and Mode

	Borrow mode	Direct Mode
Monochromatic display	0-1	0-1
4-plane color display	0-15	1-6
8-plane color display	0-255	1-6, 16-255

14.1.2. Changing the Color Map

Functions:

```
GM_$DISPLAY_SET_COLOR_MAP
GM_$DISPLAY_INQ_COLOR_MAP
```

GM_$DISPLAY_SET_COLOR_MAP changes a specified number of values in the color map.

GM_$DISPLAY_INQ_COLOR_MAP retrieves a specified number of values in the color map.

14.2. Using Viewport Techniques

Functions:

```
GM_$VIEWPORT_SET_BACKGROUND_VALUE
GM_$VIEWPORT_INQ_BACKGROUND_VALUE
```

As discussed in Section 13.9, attribute blocks can be assigned to attribute classes for individual viewports. The attribute class assignment can include a plane mask. If the plane mask assigned to attribute class 1 does not enable all planes in the GM bitmap, the following occurs. Not all planes are cleared before displaying into that viewport. (This is true for borrow, direct, and main-bitmap modes). You can use this characteristic to include a background grid that is not constantly erased and redrawn. This technique also makes it possible to superimpose two segments into the same viewport.

The following describes some additional techniques to use with viewports.

GM_$VIEWPORT_SET_BORDER_SIZE sets the border size of the current viewport to the specified values, either in pixels or in fraction-of-bitmap coordinates. This routine sets sizes of the four edges independently, for each viewport.

The default border type is in pixels, and the default width is 1,1,1,1. Viewport borders are drawn with color value 1 for compatibility with monochrome nodes. Also for this compatibility, the 2D GMR package sets the color map for color value 1 to white.

With a color node, you may want to use the viewport background color to differentiate viewports from the overall display or the window background. Changing the color map to black is usually not practical because the cursor is also set to color value 1. An alternative is to create the viewport, set the border width to 0 pixels, and then refresh the viewport.

To change the viewport background value, you can use a procedure like the following:

- Assign a background pixel value for each viewport. The default is 0. The pixel value displayed is affected by the plane mask assigned to attribute class 1.

- Change the color by changing the color map. The default fill and text background values are -2. This sets them equal to the viewport background value.

GM_$VIEWPORT_SET_BACKGROUND_VALUE sets the pixel value used for the background of the specified viewport. GM_$VIEWPORT_INQ_BACKGROUND_VALUE returns the pixel value set for the background of the specified viewport.

14.3. Program with Advanced Viewing Techniques

The following program changes the color map values; assigns a plane mask to viewports; displays a grid; changes the plane mask; assigns viewport background values; displays segments in more than one viewport; closes the file; and terminates the package.

```
PROGRAM course6;

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';
%INCLUDE '/sys/ins/gmr.ins.pas';
%INCLUDE '/sys/ins/pfm.ins.pas';
%LIST;

CONST

    gm_default_aclass = 1;

    repeats      = 10;
    space        = 25;
    one_second   = 250000;

VAR

    bitmap_size           : gm_$point16_t;
    st                    : status_$t;

    b                     : gm_$boundsreal_t;
    vpid2                 ,
    vpid3                 ,
    vpid4                 : INTEGER;

    ablockid              : INTEGER;
    k                     : INTEGER;
    m                     ,
    n                     : INTEGER;
    color_array           : ARRAY [ 8 .. 15 ] OF gm_$color_entry_t;

    file_id               : INTEGER;
    sid1                  ,
    sid2                  ,
    sid3                  ,
    sid4                  : gm_$segment_id_t;
    pt1                   ,
    pt2                   ,
    transl                : gm_$point16_t;
    rtransl               : gm_$pointreal_t;

    PROCEDURE check
        ( IN status : status_$t
        );
    BEGIN
        IF status.all <> status_$ok
        THEN pfm_$error_trap( status );
        END;
    PROCEDURE wait
        ;
    VAR
        pause    : time_$clock_t;
        status   : status_$t;
    BEGIN
        pause.low32 := 5 * one_second;
        pause.high16 := 0;
        time_$wait                  { Wait five seconds. }
            ( time_$relative
            , pause
            , status
            );
        check( status );
        END;

BEGIN

    bitmap_size.x := 1024;
    bitmap_size.y := 1024;

    gm_$init                        { Initialize the 2D GMR package. }
        ( gm_$borrow
        , stream_$stdout
        , bitmap_size
        , 8
        , st
        );

    gm_$file_create                 { Create a file. }
        ( 'gmfile'
        , 6
        , gm_$overwrite
        , gm_$1w
        , file_id
        , st
        );

    gm_$segment_create              { Create segment 'grid.' }
        ( 'grid'
        , 4
        , sid1
        , st
        );

    pt1.x := 0;                     { 'Grid' points are zero-sized rectangles. }
    FOR m := 1 TO 8
    DO BEGIN
        pt1.x := pt1.x + 100;
        pt1.y := 0;
        FOR n := 1 TO 8
        DO BEGIN
            pt1.y := pt1.y + 100;
            gm_$rectangle_16
                ( pt1
                , pt1
                , FALSE
                , st
                );
        END;
    END;

gm_$segment_close               { Close segment 'grid.' }
    ( TRUE
    , st
    );

b.xmin := 0.0;
b.ymin := 0.0;
b.xmax := 0.49;
b.ymax := 0.49;
gm_$viewport_set_bounds         { Shrink viewport 1. }
    ( b
    , st
    );

b.xmin := 0.51;
b.ymin := 0.0;
b.xmax := 1.0;
b.ymax := 0.49;
gm_$viewport_create             { Create viewport 2. }
    ( b
    , vpid2
    , st
    );

b.xmin := 0.0;
b.ymin := 0.51;
b.xmax := 0.49;
b.ymax := 1.0;
gm_$viewport_create             { Create viewport 3. }
    ( b
    , vpid3
    , st
    );

b.xmin := 0.51;
b.ymin := 0.51;
b.xmax := 1.0;
b.ymax := 1.0;
gm_$viewport_create             { Create viewport 4. }
    ( b
    , vpid4
    , st
    );

FOR k := 8 TO 15                { Red + green = yellow. }
DO WITH color_array[ k ]
DO BEGIN
    red   := 1.0;
    green := 1.0;
    blue  := 0.0;
    END;

gm_$display_set_color_map       { Set color values 8 to 15 to yellow. }
    ( 8
    , 8
    , color_array
    , st
    );

gm_$ablock_create              { Create an ablock. }
    ( 1
    , ablockid
    , st
    );

gm_$ablock_set_draw_value      { For the ablock, set the draw value to 9. }
    ( ablockid
    , 9
    , st
    );

gm_$ablock_assign_display      { Assign the ablock to the default aclass. }
    ( gm_default_aclass
    , ablockid
    , st
    );

gm_$display_file               { Display 'grid' in viewport 4. }
    ( st
    );

wait;                         { Wait a moment. }

gm_$ablock_copy               { Reset the ablock to default attributes. }
    ( 1
    , ablockid
    , st
    );

gm_$ablock_set_plane_mask    { For the ablock, set plane mask to [0,1,2]. }
    ( ablockid
    , TRUE
    , [ 0 .. 2 ]
    , st
    );

gm_$segment_create            { Create segment 'box.' }
    ( 'box'
    , 3
    , sid2
    , st
    );

pt1.x := 0;
pt1.y := 0;
pt2.x := 10;
pt2.y := 10;
gm_$rectangle_16              { Add a rectangle to 'box.' }
    ( pt1
    , pt2
    , FALSE
    , st
    );

gm_$segment_close             { Close segment 'box.' }
```

```
    ( TRUE
    , st
    );

gm_$segment_create              { Create segment 'row.' }
    ( 'row'
    , 3
    , sid3
    , st
    );

transl.y := 0;                  { Instance segment 'box' into segment 'row.'}
transl.x := 0;
FOR k := 1 TO repeats
DO BEGIN
    transl.x := transl.x + space;
    gm_$instance_translate_2d16
        ( sid2
        , transl
        , st
        );
    END;

gm_$segment_close               { Close segment 'row.' }
    ( TRUE
    , st
    );

gm_$segment_create                    { Create segment 'block.' }
    ( 'block'
    , 5
    , sid4
    , st
    );

transl.y := 50;              { Instance segment 'row' into segment 'block.' }
FOR k := 1 TO repeats
DO BEGIN
    transl.x := k ;
    transl.y := transl.y - space;
    gm_$instance_translate_2d16
        ( sid3
        , transl
        , st
        );
    END;

gm_$segment_close                { Close segment 'block.' }
    ( TRUE
    , st
    );

gm_$display_segment              { Display segment 'block' in viewport 3. }
    ( sid4
    , st
    );

wait;                           { Wait a moment. }
```

```
rtransl.x := 0.5;
rtransl.y := 1.0;
gm_$view_scale                  { For viewport 3, zoom out. }
    ( 0.25
    , rtransl
    , st
    );

rtransl.x := -0.06;
rtransl.y := 0.0;
FOR k := 1 TO 5
DO gm_$view_translate           { For viewport 3, pan from left to right. }
    ( rtransl
    , st
    );

rtransl.x := 0.5;
rtransl.y := 0.5;
FOR k := 1 TO 5
DO gm_$view_scale               { For viewport 3, pan diagonally towards }
    ( 0.85                      { the lower left. }
    , rtransl
    , st
    );

wait;                               { Wait a moment. }

gm_$viewport_set_background_value   { For viewport 2, set the }
    ( vpid2                         {  background value. }
    , 2
    , st
    );

gm_$viewport_select                 { Select viewport 2. }
    ( vpid2
    , st
    );

gm_$display_segment              { Display segment 'row' in viewport 2. }
    ( sid3
    , st
    );

wait;                               { Wait a moment. }

gm_$viewport_set_background_value   { For viewport 3, set background }
    ( vpid3                         { value to 2. }
    , 3
    , st
    );

gm_$viewport_select              { Select viewport 3. }
    ( vpid3
    , st
    );

gm_$display_segment              { Display segment 'block' in viewport 3. }
    ( sid4
    , st
    );

wait;                             { Wait a moment. }

gm_$file_close                    { Close the file. }
    ( TRUE
    , st
    );

gm_$terminate                     { Terminate the 2D GMR package. }
    ( st
    );

END.
```

Chapter 15 Programming Techniques

This chapter presents techniques for using tags and for optimizing performance when you use the 2D GMR package. Some of the relationships of the DOMAIN Core Graphics, the DOMAIN Graphics Primitives package and the DOMAIN 2D GMR Resource package are discussed.

15.1. Using Tags

Functions:

GM_$TAG
GM_$TAG_LOCATE

Tags provide a mechanism to access the database at a particular place. For example, you may have another database running alongside the 2D GMR package. You can use a tag to flag a place in a segment for accessing information in another database.

GM_$TAG inserts a comment into the metafile.

GM_$TAG_LOCATE locates a comment within a specified range of segments in the current metafile. The routine returns the identification of the lowest-numbered segment that the comment is found in.

This routine uses the wildcard options of the command line parser. For a description of the command line parser, see the *DOMAIN System Command Reference.*

15.2. Program Technique: Using Tags

Tags are especially usful with large metfiles. The following program fragment illustrates the use of tags.

```
.
.
.

{ The current segment is 'assembly_sid.'  A part is instanced into
  the assembly, followed by a tag with the part's unique part number. }

gm_$instance_translate_2d16
    ( part_sid                  { Segment defining part }
    , location                  { Location of part }
    , status
    );

gm_$tag
    ( part_number               { ASCII string with part number }
    , part_number_len           { Length of ASCII string }
```

```
        , status
        );

    .
    .
    .

    { Later, the user needs to find the part with the given part number: }

    gm_$tag_locate
        ( part_number                 { ASCII string with part number }
        , part_number_len             { Length of ASCII string }
        , 0                           { Lacking more specific information, }
        , gm_$max_segment - 1         {       search all segments. }
        , assembly_sid                { Output the segment id containing the
        , status                        part. }
        );

    .
    .
    .
```

15.3. Optimizing Performance

This section presents some techniques for optimizing performance in the applications you build with the 2D GMR package.

15.3.1. Sorting by Location in the Picture

The 2D GMR package keeps track of the rectangular bounds of each segment in a segment header attached to the segment data. When displaying a file, the 2D GMR package can therefore rapidly reject segments that do not overlap the current viewport, without examining individual commands within those segments.

You can, therefore, improve performance by sorting order-independent commands into segments based on their location in the picture. By grouping commands into segments with relatively small bounds, you can increase the number of segments that can be completely ignored during display.

The 2D GMR package cannot sort data for you because the package assumes that commands must be executed in the order you specify.

15.3.2. Segment Size

For files with many (hundreds or thousands) of segements, use unnamed segments. This avoids the overhead of checking for duplication of names.

15.3.3. Rectangles and Rotations

If you regularly apply rotations other than 90 or 180 degrees to rectangles, use the GM_$POLYLINE... routines, not the GM_$RECTANGLE... routines. With rectangle routines, the picture is not drawn incorrectly; however, the picture is displayed faster with polyline commands.

15.3.4. Compacting Files

To reduce storage space for old files, you can develop a compacting utility using GM_$FILE_COMPACT. For a description of this routine and an example of such a utility, see GM_$FILE_COMPACT in *DOMAIN System Call Reference*, Volume 1.

15.3.5. Releasing and Acquiring the Display

The listing of insert files at the top of the program "hotel.gm" in Appendix D includes gpr.ins.pas. This file gives access to the DOMAIN Graphics Primitives (GPR) routines. In general, the mixing of 2D GMR and GPR is not recommended unless you specify within-GPR mode with the routine GM_$INIT. Here the GPR routine provides the best way to release the display that 2D GMR must acquire. The GPR routine releases the display to allow writing output to a stream.

Mixing 2D GMR and GPR calls in other ways or in other contexts is *not* recommended or supported.

15.3.6. Long Identifiers

In C and FORTRAN, identifiers may be no longer than a maximum of 32 characters. In C programs, the compiler truncates the name to 32 characters. In FORTRAN programs, you need to shorten the following routine names to 32 characters as illustrated:

```
                                 |
12345678901234567890123456789012 | 34567890
---------------------------------|---------
GM_$ABLOCK_INQ_FILL_BACKGROUND_V | ALUE
GM_$ABLOCK_INQ_TEXT_BACKGROUND_V | ALUE
GM_$ABLOCK_SET_FILL_BACKGROUND_V | ALUE
GM_$ABLOCK_SET_TEXT_BACKGROUND_V | ALUE
GM_$INQ_INSTANCE_TRANSFORM_2DREA | L
GM_$INQ_INSTANCE_TRANSLATE_2DREA | L
GM_$VIEWPORT_INQ_BACKGROUND_VALU | E
GM_$VIEWPORT_SET_BACKGROUND_VALU | E
```

15.3.7. Color Map on Color Nodes

The 2D GMR package currently sets color 1 to white on color nodes for portability of applications developed on monochrome nodes. The viewport borders and the cursor are drawn with color 1. For nonwhite cursors and viewport boundaries on color nodes, use GM_$DISPLAY_SET_COLOR_MAP to respecify color 1.

When 2D GMR terminates, it currently resets color 1 to whatever it was when the package was initialized. This is true of color nodes only. If you use 2D GMR in borrow mode, the entire color map is reset when the packge terminates. (The resetting is not by 2D GMR, but by GPR.)

15.3.8. Fault Handlers

The 2D GMR package has its own "clean-up" handler that terminates 2D GMR whenever faults are encountered. It is not necessary for an application to install its own fault handler for this purpose. In fact, an application-installed fault handler will not work because 2D GMR will no longer be initialized by the time the fault handler is called.

15.4. For Users of Both 2D GMR and GPR

The 2D GMR functions are similar to DOMAIN Graphics Primitives (GPR) in many ways. The 2D GMR package has new routines for handling files, segments, and viewing.

There are a few major differences between 2D GMR routines and similar GPR routines. Two examples are the coordinate systems used and the use of defined points, rather than current position.

Coordinate Systems

The 2D GMR package uses the coordinate system that is standard in mathematics textbooks (+x is to the right; +y is up). This is different from GPR, which like most raster display devices uses +y as down).

Thus, if only positive coordinates are used, (0,0) is at the bottom left of the bitmap, not the top left as in GPR.

The only cases in which 2D GMR uses +y as down is in its definition of cursor patterns and in the output of GM_$PRINT... routines.

Defined Points and Current Position

The 2D GMR package requires that all coordinates be specified for each command. There is no "current position" kept from one command to the next, as there is in GPR. Thus, with the 2D GMR segment, you can insert or delete commands without affecting the interpretation of coordinate data in other commands in the segment.

Number of Planes Initialized

The input parameter to GM_$INIT is the number of planes to be initialized (that is, 1 or 8), not the number of the highest plane, as in GPR_$INIT (that is, 0 or 7).

15.5. For Previous Users of DOMAIN Core Graphics

The 2D GMR functions are similar to Core in many ways. Both allow you to define coordinates in one coordinate system and display them in another.

There are a few major differences between 2D GMR routines and similar Core routines. Two examples are the number of segments displayed and the use of defined points rather than current position.

Segments Displayed

In Core, multiple segments can be displayed in one viewport.

The 2D GMR package keeps only one segment in a viewport. If you wish to display more than one segment in a viewport, you must create a new segment and insert into it instances of all the segments you want to display.

When you use the 2D GMR package and specify within-GPR mode, you may display multiple segments in the GPR bitmap.

Defined Points and Current Position

The 2D GMR package requires that all coordinates be specified for each command. There is no "current position" kept from one command to the next, as there is in Core. Thus, with the 2D GMR segment, you can insert or delete commands without affecting the interpretation of coordinate data in other commands in the segment.

Temporary Segments

In Core, a temporary segment is put on the screen. No copy of the segment is kept.

In 2D GMR, a "temporary segment" is different. It is a segment that is stored like any other segment while the file is open; however, it is deleted when the file is closed.

Core Imaging and Viewing Transformations

In Core, you can specify a viewing transform to be applied to data that you have already supplied to the Core package. This transform is applied before the Core package stores the data as a display list. Core also provides an image transform, which is applied to stored data at display time.

2D GMR provides no mechanism analogous to the Core viewing transform described in the previous paragraph. In the 2D GMR package, the data is always stored untransformed as you supply it.

Incremental Display

In Core, every time you execute a command, the display is immediately updated.

In GM_$REFRESH_WAIT (default) or GM_$REFRESH_INHIBIT refresh states, the display is not updated each time you add data to the file. Incremental display can occur only in GM_$REFRESH_UPDATE and GM_$REFRESH_PARTIAL refresh states, and only in GM_$BORROW and GM_$DIRECT mode.

Appendix A
Glossary

Ablock	See Attribute block.
Aclass	See Attribute class.
Attribute	A characteristic of the manner in which a primitive graphic operation is to be performed (for example, line type or text value).
Attribute block	A data structure that holds a collection of values of attributes.
Attribute class	A means for referring to a collection of attribute values from within a metafile, with the particular attribute values defined elsewhere in the file or when the file is displayed.
Attribute command	A command in a metafile that affects the form in which subsequent primitive commands are to be displayed.
Bit plane	A one-bit-deep layer of a bitmap. On a monochromatic display, displayed bitmaps contain one plane. On a color display, displayed bitmaps may contain more planes, depending on the hardware configuration and the number of bits per pixel.
Bitmap	A three-dimensional array of bits having width, height, and depth. When a bitmap is displayed, it is treated as a two-dimensional array of sets of bits. The color of each displayed pixel is determined by using the set of bits in the corresponding pixel of the frame-buffer bitmap as an index into the color table.
Bitmap coordinates	Coordinates of points inside the GM bitmap, expressed as fractions of the GM bitmap. The lower left corner is referred to as (0.0, 0.0); the upper right corner as (1.0, 1.0). Note that if the GM bitmap is not square, the units in the x and y directions are different.
Borrow mode	A mode for use with the 2D GMR package whereby a program borrows the entire screen from the Display Manager and performs graphics operations by directly calling the display driver. The display is on the full screen, which is temporarily borrowed from the Display Manager.
Button	A logical input device used to provide a choice from a small set of alternatives. A physical device of this type is the selection buttons on a mouse.
Color map	See Color table.
Color table	A set of color table entries, each of which can store one color value. Each color value contains red, green, and blue component values. Each entry is accessed by a color table index.

Color table entry	One location in a color table. Each entry stores one color value that can be accessed by a corresponding color table index.
Color table index	An index to a particular color table entry.
Color value	The numeric encoding of a color. A color value is stored in a color table entry. Each color value consists of three component values: the first stores the value of the red component of the color, the second stores the value of the green component of the color, and the third stores the value of the blue component. Each component value is specified as a real number in the range 0.0 to 1.0, where 0.0 is the absence of the primary color and 1.0 is the full intensity color.
Command	A single element of a picture as stored in a metafile. Commands are categorized as primitive commands, attribute commands, instance commands, and tag commands.
Current command	The command in the current segment after which new commands are to be inserted. It is also the command that you can inquire about, replace, or delete. When you open a segment, the last command becomes the current command, allowing new commands to be appended. You may use GM_$PICK_COMMAND to change the current command.
Current file	The file currently being operated on. The current file can be changed by selecting another previously opened file or by opening (creating) an additional file.
Current picked segment	The segment selected by the pick-segment operation. It is used as a base for further pick-segment operations.
Current segment	The segment currently open for editing.
Current viewport	The currently selected viewport. The current viewport can be changed by selecting another existing viewport or by creating a new viewport.
Direct mode	A mode for use of the 2D GMR package whereby the package performs graphics operations in a window borrowed from the Display Manager. Direct mode allows graphics programs to coexist with other activities on the screen.
Display	The entire monitor screen.
Display Manager	The program that manages the display and allocates Display Manager windows.
Display Manager window	One section of the display, provided by the Display Manager. This window does not include the edges reserved by the Display Manager.
File	See Metafile.

Font	One set of alphanumeric and special characters. The font in which text is to be displayed is determined by the package using the user-selected font family and text size attributes.
Font family	A list of similar fonts of differing size. The 2D GMR user creates an ASCII file containing this list.
Font family file	An ASCII file listing the fonts in the font family, one font per line.
GM bitmap	The bitmap in which the 2D GMR package is initialized. In direct mode, this is part of the Display Manager window in which the package was initialized. In borrow mode, this is the entire current display. In main-bitmap mode, this is a main-memory bitmap.
Input device	A device such as a function key, touchpad, or mouse that enables a user to provide input to a program.
Input event	An input primitive that is created by a user's interaction with a device such as a keyboard, button, mouse, or touchpad.
Instance command	A command in a metafile that calls for another segment to be displayed, with a particular transformation applied. This is similar to a subroutine call.
Instanced segment	The segment referred to by an instance command in another segment.
Instancing segment	The segment that contains the instance command that refers to the instanced segment.
Keyboard	A logical input device used to provide character or text string input. One physical device of this type is the alphanumeric keyboard.
Line style	An attribute that specifies the style of lines and polylines (for example, solid or dotted).
List of picked segments	The linked sequence of instancing and instanced segments selected by a series of pick-segment operations, starting with the viewport primary segment (or primary segment in no-bitmap mode) and ending with the current picked segment.
Locator	A logical input device used to specify one position in coordinate space (for example, a touchpad, data tablet, or mouse).
Logical input device	An abstraction that refers to any of a group of physical input devices that provide similar input data. For example, the logical input device "button" can refer to physical buttons on a mouse, or to physical buttons on a data tablet puck.

Main-bitmap mode	A mode for use with the 2D GMR package whereby a program runs in a bitmap allocated in main memory, without using the display.
Metafile	A device-independent collection of picture data that can be displayed. (Also referred to as a file.)
Mode	One of four modes for use of the 2D GMR package, selected when the 2D GMR package is initialized. See Borrow mode, Direct mode, Main-bitmap mode, and No-bitmap mode.
Modeling routines	Graphics metafile routines used to insert commands into metafiles or to edit metafiles.
No-bitmap mode	A mode for use with the 2D GMR package whereby a program runs without a main-memory or display bitmap. Viewing routines may not be performed in this mode.
Open file	Any of the files that have been opened during this session and have not yet been closed. More than one file may be open at one time.
Pick aperture	The region in segment coordinate space within which pick routines will search for commands and segments.
Pick mask	A number that is compared bit by bit with a segment's pickable value to determine if the segment is pickable. If any bit is "1" in both the segment's pickable number and the pick mask, the segment may be picked (see also Pick threshold and Pick aperture). If not, the segment is not picked.
Pick operation	The process of selecting commands or segments.
Pick threshold	A number that is compared to a segment's pickable value to determine if the segment is pickable. If the segment's pickable number is greater than or equal to the pick threshold, the segment may be picked (see also Pick mask and Pick aperture). If not, the segment is not picked.
Pickable value	A number assigned to each segment that is used to determine whether a segment is to be considered during pick-segment operations.
Picture	The entire contents of a file as drawn; it may be larger or smaller than either the GM bitmap or viewport.
Picture element	A single element of a two-dimensional displayed image or of a two-dimensional location within a bitmap. It is commonly called a pixel.
Pixel	See Picture element.
Pixel value	The set of bits at a two-dimensional location within a bitmap. A pixel value is used as an index to the color map.
Polyline	A linked set of line segments.

Primary segment	The segment that is the logical start of the file for display purposes. The routine GM_$DISPLAY_FILE assumes that you wish to display this segment and all of the segments that it instances.
Primitive command	A command in a metafile that describes a single least divisible graphic operation of a stored picture (for example, lines, polylines, and text). See also User-defined primitive.
Routine	One of the procedures or functions of the 2D GMR package. Routines are categorized as modeling routines and viewing routines.
Scan line	A row of pixels; one horizontal line of a bitmap.
Segment	A collection of commands in the metafile that can be referred to as a group. See also current segment.
Segment-exponent format	A format for storage of coordinate data, similar to floating point. Instead of an exponent and a mantissa for each coordinate, there is one exponent for the entire segment and a separate 16-bit or 32-bit mantissa for each coordinate.
Tag command	A command in a metafile that contains a comment. The comment data can be retrieved by the user, but is ignored when the file is displayed.
User-defined Primitive	A primitive routine, exclusive to within-GPR mode. This type of routine is defined and specified by the user.
View	The part of a picture that is currently seen through a viewport. For example, translating or scaling a view affects what is visible through the viewport.
Viewing routines	Graphics metafile routines used to control the form in which metafiles are displayed.
Viewport	All or part of the window, excluding its border if one exists. The viewport is the physical "hole" in the window through which graphic output or other processes are visible. Moving the viewport within the GM bitmap does not scale the view.
Viewport primary segment	The segment currently displayed within a viewport. Other segments that are instanced directly or indirectly by this segment may also be displayed.

Appendix B
Keyboard Charts

The following two charts and figures give the 8-bit ASCII values generated for two DOMAIN keyboards: 880 and low-profile. These charts include characters used in keystroke events. The columns represent the four highest-order bits of an 8-bit value. The rows represent the four lowest-order bits of an 8-bit value. For a more complete description of conventions for naming keys, see the *DOMAIN System Command Reference*.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	^SP	^P	SP	0	@	P	`	p		R1		R1U	F1	F1S	F1U	F1C
1	^A	^Q	!	1	A	Q	a	q	L1	R2	L1U	R2U	F2	F2S	F2U	F2C
2	^B	^R	"	2	B	R	b	r	L2	R3	L2U	R3U	F3	F3S	F3U	F3C
3	^C	^S	#	3	C	S	c	s	L3	R4	L3U	R4U	F4	F4S	F4U	F4C
4	^D	^T	$	4	D	T	d	t	L4	R5	L4U	R5U	F5	F5S	F5U	F5C
5	^E	^U	%	5	E	U	e	u	L5	BS	L5U	R2S	F6	F6S	F6U	F6C
6	^F	^V	&	6	F	V	f	v	L6	CR	L6U	R3S	F7	F7S	F7U	F7C
7	^G	^W	'	7	G	W	g	w	L7	TAB	L7U	R4S	F8	F8S	F8U	F8C
8	^H	^X	(	8	H	X	h	x	L8	STAB	L8U	R5S	R1S	L8S	L1A	L1AU
9	^I	^Y	)	9	I	Y	i	y	L9	CTAB	L9U		L1S	L9S	L2A	L2AU
A	^J	^Z	*	:	J	Z	j	z	LA		LAU		L2S	LAS	L3A	L3AU
B	^K	ESC	+	;	K	[	k	{	LB		LBU		L3S	LBS	R6	R6U
C	^L	^\	,	<	L	\	l	|	LC		LCU		L4S	LCS	L1AS	
D	^M	^]	-	=	M	]	m	}	LD		LDU		L5S	LDS	L2AS	
E	^N	^~	.	>	N	^	n		LE		LEU		L6S	LES	L3AS	
F	^O	^?	/	?	O		o	DEL	LF		LFU		L7S	LFS	R6S	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure B-1. Low-Profile Keyboard Chart - Translated User Mode

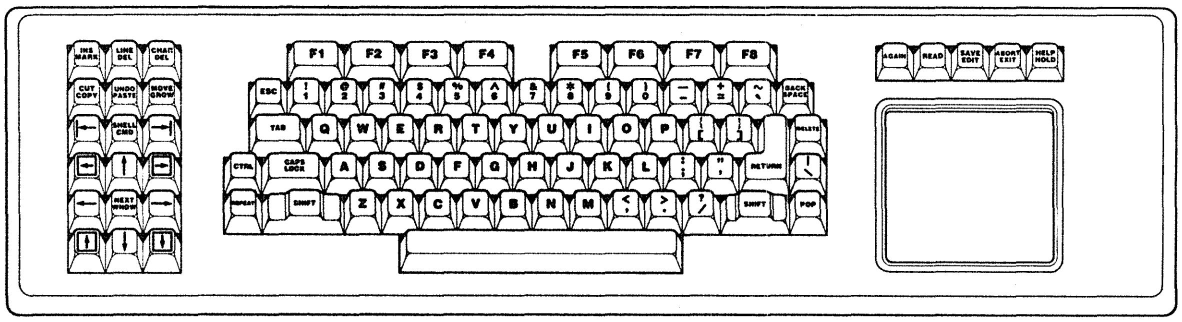


Figure B-2. Low-Profile Keyboard

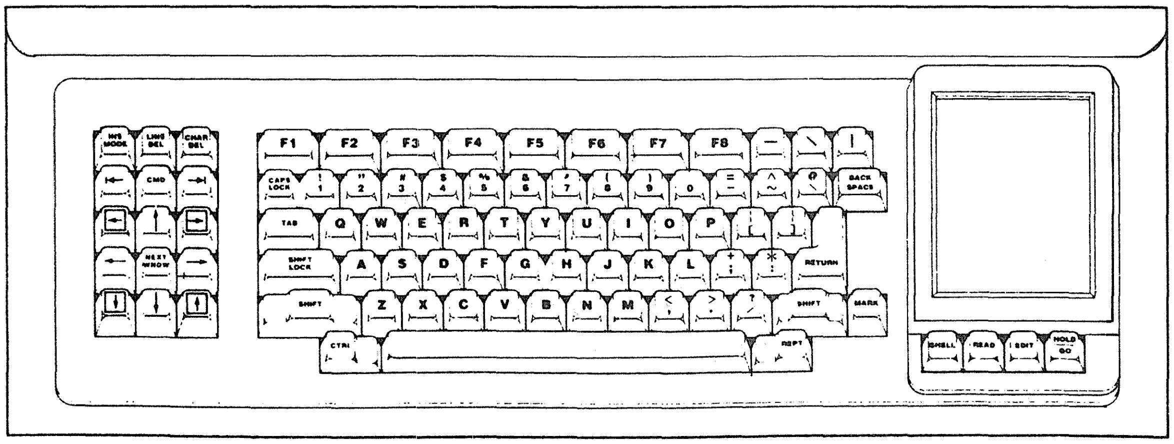


Figure B-3. **880 Keyboard**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	^`	^P	SP	0	@	P	`	p		R1		R1U	F1	F1S	F1U	F1C
1	^A	^Q	!	1	A	Q	a	q	L1	R2	L1U	R2U	F2	F2S	F2U	F2C
2	^B	^R	"	2	B	R	b	r	L2	R3	L2U	R3U	F3	F3S	F3U	F3C
3	^C	^S	#	3	C	S	c	s	L3	R4	L3U	R4U	F4	F4S	F4U	F4C
4	^D	^T	$	4	D	T	d	t	L4	R5	L4U	R5U	F5	F5S	F5U	F5C
5	^E	^U	%	5	E	U	e	u	L5	BS	L5U		F6	F6S	F6U	F6C
6	^F	^V	&	6	F	V	f	v	L6	CR	L6U		F7	F7S	F7U	F7C
7	^G	^W	'	7	G	W	g	w	L7	TAB	L7U		F8	F8S	F8U	F8C
8	^H	^X	(	8	H	X	h	x	L8	STAB	L8U		N0	N8	N0U	N8U
9	^I	^Y	)	9	I	Y	i	y	L9	CTAB	L9U		N1	N9	N1U	N9U
A	^J	^Z	*	:	J	Z	j	z	LA		LAU		N2	N.	N2U	N.U
B	^K	^[	+	;	K	[	k	{	LB		LBU		N3	N=	N3U	N=U
C	^L	^\	,	<	L	\	l	|	LC		LCU		N4	N+	N4U	N+U
D	^M	^]	-	=	M	]	m	}	LD		LDU		N5	N-	N5U	N-U
E	^N	^~	.	>	N	^	n	~	LE		LEU		N6	N*	N6U	N*U
F	^O	^/	/	?	O		o	^	LF		LFU		N7	N/	N7U	N/U
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Figure B-4. 880 Keyboard Chart - Translated User Mode

Appendix C
Program to Read the Contents of a Metafile

The following program prints out the entire contents of a metafile in a form you can read.

```
{ tread.pas }

program tread;
%nolist;
%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/gmr.ins.pas';
%list;

VAR
   st : status_$t;
   name : array [1..100] of char;
   lname : integer;
   file_id : integer;

   num_seg, max_seg_id, seg, seg_id : gm_$segment_id_t;
   ctype : gm_$command_type_t;
   dtype : gm_$data_type_t;
   i,j : integer;
   k : integer32;
   r : real;
   pt1, pt2 : gm_$point16_t;
   rpt1, rpt2 : gm_$pointreal_t;
   rot : gm_$rotate_real2x2_t;
   pt : gm_$point_array16_t;
   rpt : gm_$point_arrayreal_t;

   fill, close : boolean;
   style : gm_$line_style_t;
   pattern : gm_$draw_pattern_t;
   mask : gm_$plane_mask_t;
   rotate : real;



procedure check;    { internal }

    begin
    if ( st.all <> status_$ok ) then
        error_$print (st);
    end; { procedure check }


BEGIN
gm_$init(gm_$no_bitmap,1,pt1,1,st); check;
gm_$file_open('gmfile',6,gm_$wr,gm_$1w,file_id,st); check;

gm_$segment_inq_count(num_seg, max_seg_id, st); check;
for seg := 0 to max_seg_id do
   begin
```

```
gm_$segment_inq_name(seg,name,lname,k,st);
if ( st.all = gm_$segment_id_invalid ) then
   NEXT;
check;
writeln;
writeln('      segment', seg, '  ', name:lname);

gm_$segment_open(seg,st); check;
gm_$pick_command(gm_$start,st); check;
gm_$pick_command(gm_$step,st); check;
while ( st.all = 0 ) do
   begin
   gm_$inq_command_type(ctype, dtype, st); check;
   case ( ctype ) of
      gm_$taclass :
         begin
         gm_$inq_aclass(i,st);
         writeln('aclass ', i);
         end;
      gm_$tcircle_2d :
         begin
         if ( dtype = gm_$16 ) then
            begin
            gm_$inq_circle_16(pt1,i,fill,st); check;
            writeln('circle 16      ', pt1.x, pt1.y);
            writeln('               ', i);
            end
         else if ( dtype = gm_$32 ) then
            begin
            gm_$inq_circle_real(rpt1,r,fill,st); check;
            writeln('circle real    ', rpt1.x, rpt1.y);
            writeln('               ', r);
            end;
         if ( fill ) then
            writeln('               filled')
         else
            writeln('               not filled');
         end;
      gm_$tdraw_raster_op :
         begin
         gm_$inq_draw_raster_op(i,st); check;
         writeln('draw raster op ', i);
         end;
      gm_$tdrawstyle :
         begin
         gm_$inq_draw_style(style,i,pattern,j,st); check;
         writeln('draw style ', style, i);
         end;
      gm_$tdrawvalue :
         begin
         gm_$inq_draw_value(k,st); check;
         writeln('draw value ', k);
         end;
      gm_$tfillbvalue :
         begin
         gm_$inq_fill_background_value(k,st); check;
         writeln('fill background value ', k);
         end;
      gm_$tfillvalue :
```

```
   begin
   gm_$inq_fill_value(k,st); check;
   writeln('fill value ', k);
   end;
gm_$tfontfamily :
   begin
   gm_$inq_font_family(i,st); check;
   writeln('font family ', i);
   end;
gm_$tinstance_scale_2d :
   begin
   if ( dtype = gm_$16 ) then
      begin
      gm_$inq_instance_scale_2d16(seg_id,r,pt1,st); check;
      writeln('instance scale 2d16 ', seg_id, r);
      writeln('                    ', pt1.x, pt1.y);
      end
   else if ( dtype = gm_$32 ) then
      begin
      gm_$inq_instance_scale_2dreal(seg_id,r,rpt1,st); check;
      writeln('instance scale 2dreal ', seg_id, r);
      writeln('                      ', rpt1.x, rpt1.y);
      end;
   end;
gm_$tinstance_trans_2d :
   begin
   if ( dtype = gm_$16 ) then
      begin
      gm_$inq_instance_translate_2d16(seg_id,pt1,st); check;
      writeln('instance translate 2d16 ', seg_id);
      writeln('                        ', pt1.x, pt1.y);
      end
   else if ( dtype = gm_$32 ) then
      begin
      gm_$inq_instance_translate_2dreal(seg_id,rpt1,st); check;
      writeln('instance translate 2dreal ', seg_id);
      writeln('                          ', rpt1.x, rpt1.y);
      end;
   end;
gm_$tinstance_transform_2d :
   begin
   if ( dtype = gm_$16 ) then
      begin
      gm_$inq_instance_transform_2d16(seg_id,rot,pt1,st); check;
      writeln('instance transform 2d16 ', seg_id);
      writeln('                        ', rot.xx,rot.xy,rot.yx,
                                           rot.yy);
      writeln('                        ', pt1.x, pt1.y);
      end
   else if ( dtype = gm_$32 ) then
      begin
      gm_$inq_instance_transform_2dreal(seg_id,rot,rpt1,st); check;
      writeln('instance transform 2dreal ', seg_id);
      writeln('                          ', rot.xx,rot.xy,rot.yx,
                                             rot.yy);
      writeln('                          ', rpt1.x, rpt1.y);
      end;
   end;
gm_$tplanemask :
      begin
      gm_$inq_plane_mask(mask,st); check;
      writeln('plane mask ', integer16(mask));
      end;
   gm_$tpolyline_2d :
      begin
      if ( dtype = gm_$16 ) then
         begin
         gm_$inq_polyline_2d16(i,pt,close,fill,st); check;
         writeln('polyline 2d16 ');
         for j := 1 to i do
            writeln('                  ', pt[j].x, pt[j].y);
         end
      else if ( dtype = gm_$32 ) then
         begin
         gm_$inq_polyline_2dreal(i,rpt,close,fill,st); check;
         writeln('polyline 2dreal ');
         for j := 1 to i do
            writeln('                  ', rpt[j].x, rpt[j].y);
         end;
      if ( close ) then
         write('                      closed,')
      else
         write('                      not closed,');
      if ( fill ) then
         writeln(' filled')
      else
         writeln(' not filled');
      end;
   gm_$trectangle :
      begin
      if ( dtype = gm_$16 ) then
         begin
         gm_$inq_rectangle_16(pt1,pt2,fill,st); check;
         writeln('rectangle 16 ', pt1.x, pt1.y);
         writeln('             ', pt2.x, pt2.y);
         end
      else if ( dtype = gm_$32 ) then
         begin
         gm_$inq_rectangle_real(rpt1,rpt2,fill,st); check;
         writeln('rectangle real ', rpt1.x, rpt1.y);
         writeln('               ', rpt2.x, rpt2.y);
         end;
      if ( fill ) then
         writeln('                       filled')
      else
         writeln('                       not filled');
      end;
   gm_$ttag :
      begin
      gm_$inq_tag(name,lname,st); check;
      writeln('tag ', name:lname);
      end;
   gm_$ttext_2d :
      begin
      if ( dtype = gm_$16 ) then
         begin
         gm_$inq_text_2d16(pt1,rotate,name,lname,st); check;
         writeln('text 2d16 ', pt1.x, pt1.y);
```

```
                writeln('          ', rotate, name:lname);
                end
             else if ( dtype = gm_$32 ) then
                begin
                gm_$inq_text_2dreal(rpt1,rotate,name,lname,st); check;
                writeln('text 2dreal ', rpt1.x, rpt1.y);
                writeln('          ', rotate, name:lname);
                end;
             end;
          gm_$ttextbvalue :
             begin
             gm_$inq_text_background_value(k,st); check;
             writeln('text background value ', k);
             end;
          gm_$ttextsize :
             begin
             gm_$inq_text_size(r,st); check;
             writeln('text size ', r);
             end;
          gm_$ttextvalue :
             begin
             gm_$inq_text_value(k,st); check;
             writeln('text value ', k);
             end;
          end;
       gm_$pick_command(gm_$step,st);
       end;
    gm_$segment_close(false,st); check;
    end;

 gm_$file_close(true,st); check;
 gm_$terminate(st); check;
 END.
```

Appendix D
Program: Instances and Attributes

The program in this appendix displays a file as it is being created and edited. The file creates the picture in Figure D-1.

```
PROGRAM hotel;

%NOLIST;
%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pfm.ins.pas';
%INCLUDE '/sys/ins/gmr.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';
%LIST;

CONST

    one_second = 250000;
    five_seconds = 5 * one_second;

VAR

    status          : status_$t;

    file_id         : INTEGER;
    font_file_id    : INTEGER;

    sid_scene       : gm_$segment_id_t;
    sid_door        : gm_$segment_id_t;
    sid_window      : gm_$segment_id_t;
    sid_sign        : gm_$segment_id_t;
    sid_tree        : gm_$segment_id_t;
    sid_house       : gm_$segment_id_t;

    pattern         : gm_$draw_pattern_t;
    p               : gm_$point_array16_t;
    center          : gm_$point16_t;
    radius          : INTEGER;

    i               : INTEGER;

    pause           : time_$clock_t;


    PROCEDURE check;
    BEGIN
        IF status.all <> status_$ok
        THEN pfm_$error_trap( status );
        END;


BEGIN

    p[ 1 ].x := 1024;                  { Intialize the 2D GMR package. }
```

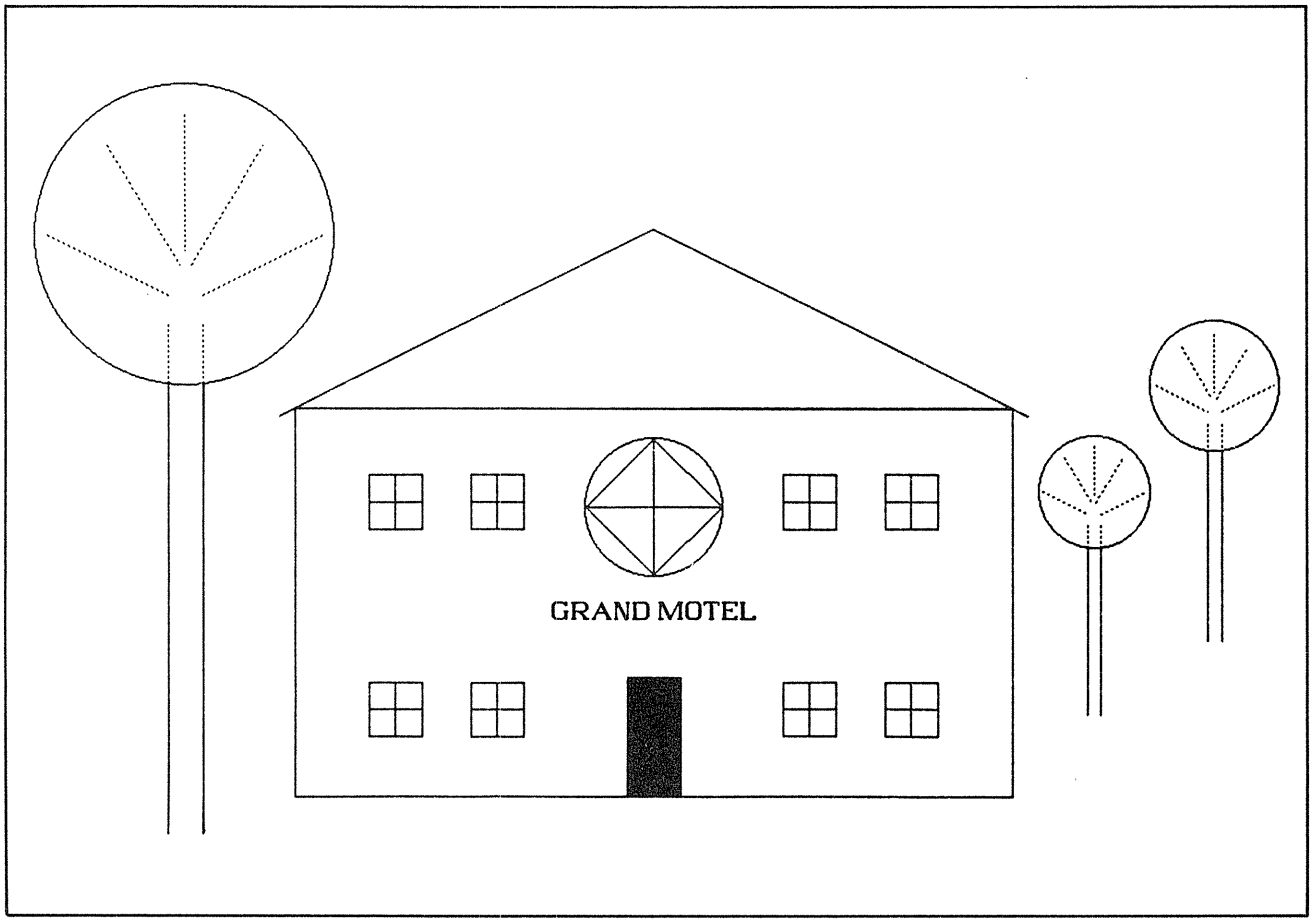


Figure D-1. A Picture Created Using Instances and Attributes

```
p[ 1 ].y := 1024;
gm_$init
    ( gm_$direct
    , stream_$stdout
    , p[ 1 ]
    , 8
    , status
    );
check;

gm_$file_create                        { Create a 2D GMR file. }
    ( 'hotel.gm'
    , 8
    , gm_$overwrite
    , gm_$1w
    , file_id
    , status
    );
check;

gm_$font_family_include                { Load a font family. }
    ( 'ff0'
    , 3
    , gm_$pixel
    , font_file_id
    , status
    );
check;

gm_$data_coerce_set_real               { Set the data coerce function. }
    ( gm_$32
    , status
    );
check;

gm_$segment_create                     { Create the segment for the door. }
    ( ''
    , 0
    , sid_door
    , status
    );
check;

p[ 1 ].x := 0;                          { Construct the door. }
p[ 1 ].y := 0;
p[ 2 ].x := 36;
p[ 2 ].y := 80;
gm_$rectangle_16
    ( p[ 1 ]
    , p[ 2 ]
    , TRUE
    , status
    );
check;

gm_$fill_value                         { Construct the door knob. }
    ( 0
    , status
    );
```

```
check;

p[ 1 ].x := 30;
p[ 1 ].y := 38;
p[ 2 ].x := 33;
p[ 2 ].y := 41;
gm_$rectangle_16
    ( p[ 1 ]
    , p[ 2 ]
    , TRUE
    , status
    );
check;

gm_$segment_close
    ( TRUE
    , status
    );
check;

gm_$segment_create                { Create the segment for the windows. }
    ( ''
    , 0
    , sid_window
    , status
    );
check;

p[ 1 ].x := 0;
p[ 1 ].y := 0;
p[ 2 ].x := 36;
p[ 2 ].y := 36;
gm_$rectangle_16
    ( p[ 1 ]
    , p[ 2 ]
    , FALSE
    , status
    );
check;

p[ 1 ].x := 0;
p[ 1 ].y := 18;
p[ 2 ].x := 36;
p[ 2 ].y := 18;
gm_$polyline_2d16
    ( 2
    , p
    , FALSE
    , FALSE
    , status
    );
check;

p[ 1 ].x := 18;
p[ 1 ].y := 0;
p[ 2 ].x := 18;
p[ 2 ].y := 36;
gm_$polyline_2d16
    ( 2
    , p
    , FALSE
    , FALSE
    , status
    );
check;

gm_$segment_close
    ( TRUE
    , status
    );
check;

gm_$segment_create                   { Create the segment for the sign. }
    ( ''
    , 0
    , sid_sign
    , status
    );
check;

gm_$text_size
    ( 14.0
    , status
    );
check;

p[ 1 ].x := 0;
p[ 1 ].y := 0;
gm_$text_2d16
    ( p[ 1 ]
    , 0.0
    , 'GRAND MOTEL'
    , 11
    , status
    );
check;

gm_$segment_close
    ( TRUE
    , status
    );
check;

gm_$segment_create                   { Create the segment for the house. }
    ( ''
    , 0
    , sid_house
    , status
    );
check;

p[ 1 ].x := 0;                        { Build the house. }
p[ 1 ].y := 0;
p[ 2 ].x := 480;
p[ 2 ].y := 260;
gm_$rectangle_16
    ( p[ 1 ]
    , p[ 2 ]
```

```
        , FALSE
        , status
        );
    check;

    p[ 1 ].x := -10;                    { Build the roof. }
    p[ 1 ].y := 255;
    p[ 2 ].x := 240;
    p[ 2 ].y := 380;
    p[3].x := 490;
    p[3].y := 255;
    gm_$polyline_2d16
        ( 3
        , p
        , FALSE
        , FALSE
        , status
        );
    check;

    p[ 1 ].x := 300;                    { Build the chimney. }
    p[ 1 ].y := 350;
    p[ 2 ].x := 300;
    p[ 2 ].y := 370;
    p[3].x := 330;
    p[3].y := 370;
    p[4].x := 330;
    p[4].y := 335;
    gm_$polyline_2d16
        ( 4
        , p
        , FALSE
        , FALSE
        , status
        );
    check;

    center.x := 240;                      { Build the round window. }
    center.y := 195;
    radius := 45;
    gm_$circle_16
        ( center
        , radius
        , FALSE
        , status
        );
    check;

    p[ 1 ].x := center.x - radius;
    p[ 1 ].y := center.y;
    p[ 2 ].x := center.x + radius;
    p[ 2 ].y := center.y;
    p[ 3 ].x := center.x;
    p[ 3 ].y := center.y - radius;
    p[ 4 ].x := center.x;
    p[ 4 ].y := center.y + radius;
    gm_$polyline_2d16
        ( 4
        , p
    , TRUE
    , FALSE
    , status
    );
check;

p[ 5 ].x := p[ 2 ].x;
p[ 5 ].y := p[ 2 ].y;
p[ 2 ].x := p[ 3 ].x;
p[ 2 ].y := p[ 3 ].y;
gm_$polyline_2d16
    ( 2
    , p
    , FALSE
    , FALSE
    , status
    );
check;

gm_$polyline_2d16
    ( 2
    , p[4]
    , FALSE
    , FALSE
    , status
    );
check;

p[ 1 ].x := 222;                    { Instance and position the door. }
p[ 1 ].y := 0;
gm_$instance_translate_2d16
    ( sid_door
    , p[ 1 ]
    , status
    );
check;

p[ 1 ].x := 50;                     { Instance and position the windows. }
p[ 1 ].y := 40;
gm_$instance_translate_2d16
    ( sid_window
    , p[ 1 ]
    , status
    );
check;

p[ 1 ].x := 118;
gm_$instance_translate_2d16
    ( sid_window
    , p[ 1 ]
    , status
    );
check;

p[ 1 ].x := 326;
gm_$instance_translate_2d16
    ( sid_window
    , p[ 1 ]
    , status
```

```
        );
    check;

    p[ 1 ].x := 394;
    gm_$instance_translate_2d16
        ( sid_window
        , p[ 1 ]
        , status
        );

    p[ 1 ].y := 180;
    gm_$instance_translate_2d16
        ( sid_window
        , p[ 1 ]
        , status
        );

    p[ 1 ].x := 326;
    gm_$instance_translate_2d16
        ( sid_window
        , p[ 1 ]
        , status
        );
    check;

    p[ 1 ].x := 118;
    gm_$instance_translate_2d16
        ( sid_window
        , p[ 1 ]
        , status
        );
    check;

    p[ 1 ].x := 50;
    gm_$instance_translate_2d16
        ( sid_window
        , p[ 1 ]
        , status
        );
    check;

    p[ 1 ].x := 172;                 { Instance and position the segment }
                                     { for the sign. }
    p[ 1 ].y := 120;
    gm_$instance_translate_2d16
        ( sid_sign
        , p[ 1 ]
        , status
        );
    check;

    gm_$segment_close
        ( TRUE
        , status
        );
    check;

    gm_$segment_create              { Create the segment for the trees. }
        ( ''
        , 0
        , sid_tree
        , status
        );
check;

p[ 1 ].x := 0;
p[ 1 ].y := 0;
p[ 2 ].x := 0;
p[ 2 ].y := 150;
gm_$polyline_2d16
        ( 2
        , p
        , FALSE
        , FALSE
        , status
        );
check;

p[ 1 ].x := 12;
p[ 2 ].x := 12;
gm_$polyline_2d16
        ( 2
        , p
        , FALSE
        , FALSE
        , status
        );
check;

p[ 1 ].x := 6;
p[ 1 ].y := 200;
gm_$circle_16
        ( p[ 1 ]
        , 50
        , FALSE
        , status
        );
check;

gm_$draw_style
        ( gm_$dotted
        , 2
        , pattern
        , 0
        , status
        );

p[ 1 ].x := 0;
p[ 1 ].y := 180;
p[ 2 ].x := -40;
p[ 2 ].y := 200;
gm_$polyline_2d16
        ( 2
        , p
        , FALSE
        , FALSE
        , status
        );
```

```
p[ 1 ].x := 12;
p[ 2 ].x := 52;
gm_$polyline_2d16
    ( 2
    , p
    , FALSE
    , FALSE
    , status
    );
check;

p[ 1 ].x := 4;
p[ 1 ].y := 190;
p[ 2 ].x := -20;
p[ 2 ].y := 230;
gm_$polyline_2d16
    ( 2
    , p
    , FALSE
    , FALSE
    , status
    );
check;

p[ 1 ].x := 8;
p[ 1 ].y := 190;
p[ 2 ].x := 32;
p[ 2 ].y := 230;
gm_$polyline_2d16
    ( 2
    , p
    , FALSE
    , FALSE
    , status
    );
check;

p[ 1 ].x := 6;
p[ 1 ].y := 195;
p[ 2 ].x := 6;
p[ 2 ].y := 240;
gm_$polyline_2d16
    ( 2
    , p
    , FALSE
    , FALSE
    , status
    );
check;

p[ 1 ].x := 0;
p[ 1 ].y := 170;
p[ 2 ].x := 0;
p[ 2 ].y := 150;
gm_$polyline_2d16
    ( 2
    , p
    , FALSE
    , FALSE
    , status
    );
check;

p[ 1 ].x := 12;
p[ 2 ].x := 12;
gm_$polyline_2d16
    ( 2
    , p
    , FALSE
    , FALSE
    , status
    );
check;

gm_$segment_close
    ( TRUE
    , status
    );
check;

gm_$segment_create                  { Create the segment called "scene." }
    ( ''
    , 0
    , sid_scene
    , status
    );
check;

p[ 1 ].x := 0;                      { Instance the segment for the house. }
p[ 1 ].y := 0;
gm_$instance_translate_2d16
    ( sid_house
    , p[ 1 ]
    , status
    );
check;

p[ 1 ].x := -85;                     { Instance, translate, and scale }
                                     { the segment for the trees. }
p[ 1 ].y := -25;
gm_$instance_scale_2d16
    ( sid_tree
    , 2.0
    , p[ 1 ]
    , status
    );

p[ 1 ].x := 530;
p[ 1 ].y := 55;
gm_$instance_scale_2d16
    ( sid_tree
    , 0.75
    , p[ 1 ]
    , status
    );
check;

p[ 1 ].x := 610;
```

```
p[ 1 ].y := 105;
gm_$instance_scale_2d16
    ( sid_tree
    , 0.85
    , p[ 1 ]
    , status
    );
check;

gm_$segment_close
    ( TRUE
    , status
    );
check;

gm_$display_segment                   { Now display the completed scene. }
    ( sid_scene
    , status
    );
check;

pause.low32  := five_seconds;         { Admire the scene for five seconds. }
pause.high16 := 0;
TIME_$WAIT
    ( time_$relative
    , pause
    , status
    );
check;

gm_$file_close                        { Close and save the file. }
    ( TRUE
    , status
    );
check;

gm_$terminate                         { Terminate the 2D GMR package. }
    ( status
    );
check;

END.
```

Appendix E
C Program Examples

This Appendix contains the programming examples presented in the manual translated into C.

A Program to Draw a Rectrangle

The following program demonstrates how to initialize the 2D GMR package, create a metafile, create a segment, and draw a rectangle (see Section 3.8 and Figure 3-3).

```
/* PROGRAM draw_rectangle */


#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gmr.ins.c"
#include "/sys/ins/time.ins.c"
#list

#define one_second     250000
#define five_seconds   (5 * one_second)
#define ten_seconds    (10 * one_second)

short              file_id;
gm_$segment_id_t   segment_id;      /* 4-byte integer              */
status_$t          st;
gm_$point16_t      pt1, pt2;        /* array of two 2-byte integers */
long               i;
gm_$point16_t      bitmap_size = {1024,1024};
time_$clock_t     pause;

main()
{
/* Define the coordinates of the rectangle to be drawn. */
    pt1.x = 100;
    pt1.y = 30;
    pt2.x = 200;
    pt2.y = 50;

/* Initialize 2D GMR. */

    gm_$init( gm_$direct,
              (short)1,
              bitmap_size,
              (short)8,
              st);

/* Create and name a metafile. */

    gm_$file_create("gmfile",
                    (short)6,
                    gm_$overwrite,
                    gm_$1w,
                    file_id,
                    st);

/* Create and name a segment. */

    gm_$segment_create("rectang_seg",
                       strlen("rectang_seg"),
                       segment_id,
                       st);

/* Insert the rectangle */
    gm_$rectangle_16( pt1,
                      pt2,
                      false,
                      st);

/* Display the file. */

    gm_$display_file(st);

/* Keep the figure displayed on the screen for five seconds.*/
    pause.low32  = five_seconds;
    pause.high16 = 0;
    time_$wait( time_$relative,
                pause,
                st );

/* Close the segment.*/
    gm_$segment_close(true,
                      st);

/* Close the metafile. */
    gm_$file_close( true,
                    st);

    gm_$terminate(st);
}
```

A Program Using Primitives and Instancing

The following program draws the design in Figure 4-2 using primitives and instancing (see Section 4.6).

```
/* PROGRAM four_rec */

#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gmr.ins.c"
#include "/sys/ins/time.ins.c"
#list

#define one_second     250000
#define five_seconds   ( 5 * one_second)
#define ten_seconds    (10 * one_second)

short                  file_id;
gm_$segment_id_t       small_id;   /* 4-byte integer               */
gm_$segment_id_t       large_id;
status_$t              st;
gm_$point16_t          pt1, pt2;   /* array of two 2-byte integers */
long                   i;
gm_$point16_t          bitmap_size = {1024,1024};
gm_$point16_t          position;   /* array of two 2-byte integers */
gm_$point_array16_t    positions;
gm_$draw_pattern_t     pattern;
time_$clock_t          pause;

main()
{

/* Initialize 2D GMR. */
    gm_$init( gm_$direct,
              (short)1,
              bitmap_size,
              (short)8,
              st);

/*Create and name a metafile.*/

    gm_$file_create("gmfile",
                    (short)6,
                    gm_$overwrite,
                    gm_$1w,
                    file_id,
                    st);

/* Create and name a segment. */

    gm_$segment_create("small_rec",
                       (short)strlen("small_rec"),
                       small_id,
                       st);
```

```
/* Define the coordinates of the rectangle. */
    pt1.x = 100;
    pt1.y = 100;
    pt2.x = 200;
    pt2.y = 200;

/* Draw one small rectangle. */

    gm_$rectangle_16( pt1,
                      pt2,
                      true,
                      st);

/* Close the segment. */

    gm_$segment_close( true,
                       st);

    gm_$segment_create("large_rec",
                        (short)strlen("large_rec"),
                        large_id,
                        st);

/* Define the coordinates of the rectangle. */
    pt1.x = 100;
    pt1.y = 100;
    pt2.x = 600;
    pt2.y = 600;

/* Draw a rectangle. */
    gm_$rectangle_16( pt1,
                      pt2,
                      false,
                      st);

/* Instance the small rectangle four times. */
    position.x = 100;
    position.y = 100;

    gm_$instance_translate_2d16( small_id,
                                 position,
                                 st);

    position.x = 300;
    position.y = 300;

    gm_$instance_translate_2d16( small_id,
                                 position,
                                 st);

    position.x = 300;
    position.y = 100;

    gm_$instance_translate_2d16( small_id,
                                 position,
                                 st);

    position.x = 100;
    position.y = 300;
    gm_$instance_translate_2d16( small_id,
                                 position,
                                 st);

/* Draw two polylines connecting four rectangles. */
    positions[0].x = 300;
    positions[0].y = 300;
    positions[1].x = 400;
    positions[1].y = 400;
    gm_$polyline_2d16(2,positions,false,false,st);

    positions[0].x = 300;
    positions[0].y = 400;
    positions[1].x = 400;
    positions[1].y = 300;

    gm_$polyline_2d16( (short)2,
                       positions,
                       false,
                       false,
                       st);

/* Close the segment. */
    gm_$segment_close( true,
                       st);

    gm_$display_segment( large_id,
                         st);

/* Keep figure displayed on the screen for five seconds. */
    pause.low32  = five_seconds;
    pause.high16 = 0;

    time_$wait( time_$relative,
                pause,
                st );

/* Close the metafile. */
    gm_$file_close( true,
                    st);

/* Terminate 2D GMR. */
    gm_$terminate(st);
}
```

A Program with Attributes and Instancing

The following program modifies the program above by adding an attribute command (see Section 5.4).

```
/* PROGRAM draw_rectangles */

#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gmr.ins.c"
#include "/sys/ins/time.ins.c"
#list

#define one_second       250000
#define five_seconds     (5 * one_second)
#define ten_seconds      (10 * one_second)

short                    file_id;
gm_$segment_id_t         small_id;   /* 4-byte integer              */
gm_$segment_id_t         large_id;
status_$t                st;
gm_$point16_t            pt1, pt2;   /* array of two 2-byte integers */
long                     i;
gm_$point16_t            bitmap_size = {1024, 1024};
gm_$point16_t            position;   /*array of two 2-byte integers  */
gm_$point_array16_t      positions;
gm_$draw_pattern_t       pattern = {'\377', '\360'};
time_$clock_t            pause;

main()
{
/* Initialize 2D GMR. */

    gm_$init( gm_$direct,
              (short)1,
              bitmap_size,
              (short)8,
              st);
/* Create and name a metafile. */

    gm_$file_create("gmfile",
                     (short)6,
                     gm_$overwrite,
                     gm_$1w,
                     file_id,st);

/* Create and name a segment. */

    gm_$segment_create("small_rec",
                        (short)strlen("small_rec"),
                        small_id,
                        st);

    gm_$draw_style(gm_$dotted,
                   (short)4,
                   pattern,
                   (short)0,
                  st);

/* Define the coordinates of the rectangle. */
    pt1.x = 100;
    pt1.y = 100;
    pt2.x = 200;
    pt2.y = 200;

/*Draw one small rectangle.*/
    gm_$rectangle_16( pt1,
                      pt2,
                      false,
                      st);

/* Close the segment. */
    gm_$segment_close(true,st);

    gm_$segment_create("large_rec",
                        (short)strlen("large_rec"),
                        large_id,st);

/* Instance the small rectangle four times. */
    position.x = 100;
    position.y = 100;

    gm_$instance_translate_2d16( small_id,
                                 position,
                                 st);

    position.x = 300;
    position.y = 300;

    gm_$instance_translate_2d16( small_id,
                                 position,
                                 st);

    position.x = 300;
    position.y = 100;

    gm_$instance_translate_2d16( small_id,
                                 position,
                                 st);

    position.x = 100;
    position.y = 300;

    gm_$instance_translate_2d16( small_id,
                                 position,
                                 st);

    gm_$draw_style( gm_$patterned,
                    (short)1,
                    pattern,
                    (short)16,
                    st);

/* Define the coordinates of the rectangle. */
    pt1.x = 100;
    pt1.y = 100;
    pt2.x = 600;
    pt2.y = 600;

/* Draw a rectangle. */

    gm_$rectangle_16( pt1,
                      pt2,
                      false,
                      st);

/* Draw two polylines connecting four rectangles. */
    positions[0].x = 300;
    positions[0].y = 300;
    positions[1].x = 400;
    positions[1].y = 400;

    gm_$polyline_2d16( (short)2,
                       positions,
                       false,
                       false,
                       st);

    positions[0].x = 300;
    positions[0].y = 400;
    positions[1].x = 400;
    positions[1].y = 300;

    gm_$polyline_2d16( (short)2,
                       positions,
                       false,
                       false,
                       st);

/* Close the segment. */

    gm_$segment_close( true,
                       st);

    gm_$display_segment( large_id,
                         st);

/* Keep figure displayed on the screen for five seconds. */
    pause.low32  = five_seconds;
    pause.high16 = 0;

    time_$wait( time_$relative,
                pause,
                st );

/* Close the metafile. */

    gm_$file_close( true,
                    st);

/* Terminate 2D GMR. */
    gm_$terminate(st);
}
```

A Program Including Text

The following program draws the design in Figure 6-1. This is a rectangle with horizontal and vertical text strings (see Section 6.5).

```
/* PROGRAM draw_rectangle_text */

#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gmr.ins.c"
#include "/sys/ins/time.ins.c"
#list

#define one_second      250000
#define five_seconds    (5 * one_second)

short              file_id;
gm_$segment_id_t   segment_id;     /* 4-byte integer */
status_$t          st;
gm_$point16_t      pt1, pt2,point;   /* array of two 2-byte integers */
long               i;
gm_$point16_t      bitmap_size = {1024,1024};
short              ffid;
time_$clock_t      pause;

main()
{
/*Initialize 2D GMR.*/

    gm_$init( gm_$direct,
              (short)1,
              bitmap_size,
              (short)8,
              st);
/* Create and name a metafile. */

    gm_$file_create("gmfile",
                     (short)6,
                     gm_$overwrite,
                     gm_$1w,
                     file_id,
                     st);

/* Create and name a segment. */

    gm_$segment_create("rectang_seg",
                        (short)strlen("rectang_seg"),
                        segment_id,
                        st);

/*Load the font family.*/

    gm_$font_family_include( "font_families",
                             (short)strlen("font_families"),
                             gm_$pixel,
                             ffid,
                             st );
    gm_$text_size( (float)14.0,
                   st);
    point.x = 5;
    point.y = 510;

    gm_$text_2d16 ( point,
                   (float)0.0 ,
                   "This is the top of the rectangle." ,
                   (short)strlen("This is the top of the rectangle."),
                   st );

    point.x = 5;
    point.y = 50;

    gm_$text_2d16 ( point,
                   (float)-90.0,
                   "This is the side of the rectangle.",
                   (short)strlen("This is the side of the rectangle."),
                   st );

/* Define the coordinates of the rectangle to be drawn. */
    pt1.x = 10;
    pt1.y = 30;
    pt2.x = 400;
    pt2.y = 500;

    gm_$rectangle_16( pt1,
                      pt2,
                      false,
                      st);

/*Close the segment.*/

    gm_$segment_close( true,
                       st);


/* Display the file. */

    gm_$display_file(st);

/* Keep figure displayed on the screen for five seconds. */
    pause.low32  = five_seconds;
    pause.high16 = 0;

    time_$wait( time_$relative,
                pause,
                st );

/* Close the metafile. */

    gm_$file_close( true,
                    st);

    gm_$terminate(st);
}
```

A Program With Stroke and Pixel Fonts

The following program loads a pixel font family file and a stroke font family file and then shifts back and forth between them using an attribute block and attribute class command (see Section 6.9). The purpose of this program is to illustrate the use of the two types of text. The attribute block and attribute class command provide an easy way to change text size. For a discussion of attribute blocks and attribute classes, see Chapter 13.

```
/* PROGRAM text */

#nolist
#include <stdio.h>
#include <math.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gmr.ins.c"
#include "/sys/ins/pfm.ins.c"
#include "/sys/ins/time.ins.c"
#list

#define aclass1      1
#define second       500000
#define cos_delta    (float)cos( 0.25 )
#define sin_delta    (float)sin( 0.25 )

status_$t             status;
gm_$segment_id_t      sid_text;
gm_$segment_id_t      sid_top;
short                 file_id;
short                 ffid_pixel;
short                 ffid_stroke;
short                 ablock_id;
gm_$point16_t         p;
gm_$point16_t         q;
gm_$boundsreal_t      dbounds;
short                 i;
short                 j;
float                 text_size;
float                 text_size_delta;
gm_$rotate_real2x2_t  rotate;
gm_$point16_t         translate;
time_$clock_t         pause;
/**************************************************************/
main()
{
    p.x = 1024;
    p.y = 1024;


/* Initialize the 2D GMR package. */

    gm_$init( gm_$direct,
              (short)1,
              p,
              (short)8,
              status);
    check(status);

/* Create and name a metafile. */
```

```
    gm_$file_create( "gmfile",
                     (short)6,
                     gm_$overwrite,
                     gm_$1w,
                     file_id,
                     status);

    check( status );

/* Set the viewport refresh state. */
    gm_$viewport_set_refresh_state( gm_$refresh_wait,
                                    status);

    check(status);

/* Include a pixel font family. */
    gm_$font_family_include( "ff0",
                             (short)3,
                             gm_$pixel,
                             ffid_pixel,
                             status);
    check(status);

/* Include a stroke font family. */
    gm_$font_family_include( "ffs",
                             (short)3,
                             gm_$stroke,
                             ffid_stroke,
                             status);

    check( status );

/* Create an ablock. */
    gm_$ablock_create( (short)1,
                       ablock_id,
                       status);

    check(status);

/* Set the ablock_id = aclass1. */
    gm_$ablock_assign_display( aclass1,
                               ablock_id,
                               status);

    check(status);

/* Create a text segment. */
    gm_$segment_create( (char *)NULL,
                        (short)0,
                        sid_text,
                        status);

    check(status);

/* Add an aclass command. */
    gm_$aclass( aclass1,
                status);

    check( status );
    p.x = - 5;
    p.y = - 5;
    q.x =  5;
    q.y =  5;

/* Add a unfilled rectangle. */

    gm_$rectangle_16( p,
                      q,
                      false,
                      status);
    check(status);

    p.x =  10;
    p.y =    0;

/* Add Left to Right text. */

    gm_$text_2d16( p,
                   (float)0.0,
                   "Left to Right",
                   (short)13,
                    status);

    check( status );

    p.x =    0;
    p.y = - 10;

/* Add Top to Bottom text. */

    gm_$text_2d16( p,
                   (float)90.0,
                  "Top to Bottom",
                   (short)13,
                   status);

    check(status);

    p.x = - 10;
    p.y =    0;

/* Add Right to Left text. */

    gm_$text_2d16( p,
                   (float)180.0,
                  "Right to Left",
                   (short)13,
                   status);

    check( status );

    p.x =   0;
    p.y =  10;

/* Add Bottom to Top text. */

    gm_$text_2d16( p,
                  (float)-90.0,
                "Bottom to Top",
                 (short)13,
                status);

    check( status );


/* Close the segment. */
    gm_$segment_close( true,
                       status);

    check(status);

    dbounds.xmin = - 50.0;
    dbounds.ymin = - 50.0;
    dbounds.xmax =  50.0;
    dbounds.ymax =  50.0;

    pause.low32  = second / 4;
    pause.high16 = 0;

    text_size = 10.0;
    text_size_delta = 1.0;

/* * * Illustrate different text sizes with pixel and stroke text. * * */

    for(j=0; j<2; j++)
    {
        if(j == 1)

/* Set ablock to pixel font family. */

            gm_$ablock_set_font_family(.ablock_id,
                                        ffid_pixel,
                                        status);

        else

/* Set ablock to stroke font family. */

            gm_$ablock_set_font_family( ablock_id,
                                        ffid_stroke,
                                        status);

        check(status);

        for(i=0; i<20; i++)
        {
            if( text_size >= 10.0)
                text_size_delta = -(fabs((double)text_size_delta ));
            else
                if(text_size <= fabs((double) text_size_delta ))
                    text_size_delta =  fabs((double) text_size_delta );

            text_size += text_size_delta;


/* Change ablock text size. */
```

```
            gm_$ablock_set_text_size( ablock_id,
                                      text_size,
                                      status);

            check( status );

/* Display the file. */
            gm_$display_file_part( dbounds,
                                   status);

            check( status );

/* Admire it for a momemt. */

            time_$wait( time_$relative,
                        pause,
                        status);

            check( status );
        }/* end for i */
    }/* end for j*/

/* Create top segment. */
    gm_$segment_create( (char *)NULL,
                        (short)0,
                         sid_top,
                         status);

    check( status );

/* Identity matrix */
    rotate.xx = 1.0;
    rotate.xy = 0.0;
    rotate.yx = 0.0;
    rotate.yy = 1.0;

/* Zero translation */
    translate.x = 0;
    translate.y = 0;

/* Instance text segment into top segment. */

    gm_$instance_transform_2d16( sid_text,
                                 rotate,
                                 translate,
                                 status);

    check( status );

/* Go into replace mode. */

    gm_$modelcmd_set_mode( gm_$modelcmd_replace,
                           status);

    check( status );

/* * * Illustrate different text angles with pixel and stroke text. * * */

    for(j=0; j<2; j++)
    {
        if( j == 1)

/* Set ablock to pixel font family. */

            gm_$ablock_set_font_family( ablock_id,
                                        ffid_pixel,
                                        status);
        else

/* Set ablock to stroke font family. */

            gm_$ablock_set_font_family( ablock_id,
                                        ffid_stroke,
                                        status);

        check( status );

        for(i=0; i<40; i++)
        {

/* Increment rotation matrix. */
            rotate.xx = cos_delta * rotate.xx + sin_delta * rotate.xy;
            rotate.yx = cos_delta * rotate.yx + sin_delta * rotate.yy;
            rotate.xy = -rotate.yx;
            rotate.yy = rotate.xx;

/* Change the angle of the instance transform. */

            gm_$instance_transform_2d16( sid_text,
                                         rotate,
                                         translate,
                                         status);

            check( status );

/* Display the file. */

            gm_$display_file_part( dbounds,
                                   status);

            check( status );

/* Admire it for a moment. */

            time_$wait( time_$relative,
                        pause,
                        status);

            check( status );

        }/*end for i */
    }/* end for j */

/* Close the top segment. */

    gm_$segment_close( true,
                       status);
    check(status);

/* Close the file. */
    gm_$file_close( true,
                    status);

    check( status );


/* Terminate the 2D GMR package. */

    gm_$terminate(status);

    check( status );
}
/*****************************************************************/
check(status)

status_$t  status;
{
    if(status.all != status_$ok)
        pfm_$error_trap(status);
}
```

Program to Change the View

This program opens and displays an existing file and then changes the view scale to shrink the viewport. (For an existing file, you may use the program called "hotel" in Appendix D . As viewports may not overlap, this makes room for a second viewport. The second viewport is created, and the file is displayed in it. The view scale is then changed for the second viewport (see Section 8.9).

```
/* PROGRAM coures2 */


#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/gmr.ins.c"
#list

status_$t           status;
name_$pname_t       name;
short               length;
short               vpid2;
short               vpid3;
short               file_id;
gm_$pointreal_t     rtransl;
gm_$boundsreal_t    b;
gm_$point16_t       bitmap_size;
short               i;

main()
{
    printf("File name:  \n");
    gets(name);
    length = strlen(name);


    bitmap_size.x = 1024;
    bitmap_size.y = 1024;

    gm_$init( gm_$direct,
              stream_$stdout,
              bitmap_size,
              (short)8,
              status);

    gm_$file_open( name,
                   length,
                   gm_$r,
                   gm_$1w,
                   file_id,
                   status );

/* Now display the file. */

    gm_$display_file(status);

/* Change the view scale. */
    rtransl.x = 0.0;
    rtransl.y = 1.0;

    gm_$view_scale( (float)0.25,
                    rtransl,
                    status);

    for(i=0; i<10; i++)
        gm_$view_scale( (float)1.05,
                        rtransl,
                        status);

    rtransl.x = 0.1;
    rtransl.y = 0.8;
    for(i=0; i<10; i++)

        gm_$view_scale( (float)0.92,
                        rtransl,
                        status);

/* Shrink the viewport. */
    b.xmin = 0.0;
    b.ymin = 0.6;
    b.xmax = 0.38;
    b.ymax = 1.0;

    gm_$viewport_set_bounds( b,
                             status);

/* Create a second viewport. */
    b.xmin = 0.4;
    b.ymin = 0.0;
    b.xmax = 1.0;
    b.ymax = 0.6;

    gm_$viewport_create( b,
                         vpid2,
                         status);

/* Display file in second viewport. */
    gm_$display_file(status);

/* Change view scale in second viewport. */
    rtransl.x = 0.7;
    rtransl.y = 0.3;
    for(i=0; i<10; i++)
        gm_$view_scale( (float)1.05,
                         rtransl,
                         status);

    for(i=0; i<3; i++)
        gm_$view_scale( (float)0.85,
                        rtransl,
                        status);

/* Switch back to first viewport. */

    gm_$viewport_select( (short)1,
                         status);
    rtransl.x = 0.0;
    rtransl.y = 0.9;
    for(i=0; i<10; i++)
        gm_$view_scale( (short)1.05,
                        rtransl,
                        status );

    for(i=0; i<3; i++)
        gm_$view_scale( (float)0.9,
                         rtransl,
                         status );

/* Switch back to second viewport. */
    gm_$viewport_select( vpid2,
                         status);

/* Translate the (second) viewport. */
    rtransl.x = 0.0;
    rtransl.y = -0.4;

    gm_$view_translate( rtransl,
                        status);

/* Shrink the second viewport. */
    b.xmin = 0.4;
    b.ymin = 0.4;
    b.xmax = 1.0;
    b.ymax = 1.0;

    gm_$viewport_set_bounds( b,
                             status);

/* Create a third viewport. */
    b.xmin = 0.0;
    b.ymin = 0.0;
    b.xmax = 1.0;
    b.ymax = 0.2;

    gm_$viewport_create( b,
                         vpid3,
                         status);

/* Display file in third viewport. */
    gm_$display_file(status);

    gm_$viewport_inq_bounds( b,
                             status);

/* Change view scale in third viewport. */
    rtransl.x = 0.9;
    rtransl.y = 0.1;
    for(i=0; i<20; i++)
        gm_$view_scale( (float)0.9,
                         rtransl,
                         status);

/* Close the file. */

    gm_$file_close( true,
                        status);

    gm_$terminate(status);
}
```

An Interactive Program

The following program creates one segment containing sixteen filled polyline commands. The user can then pick and move the commands and demonstrate three model command modes (see Section 9.3).

```
/* PROGRAM star_move */

/* The following keys are enabled and perform the following actions: */

/*                                                                    */
/*   P ..... Toggle to pick/replace a command                         */
/*   Q ..... Quit                                                     */
/*  ^X ..... Abort rubberbanding (command is NOT replaced)            */


#nolist
#include <stdio.h>
#include <math.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/gmr.ins.c"
#include "/sys/ins/pfm.ins.c"
#list

#define SETSIZE     (short)256

#define ctrlx       '\030'
#define stars_x     4
#define stars_y     4

gm_$event_t          ev_type;
char                 character;
gm_$pointreal_t      bitmap_pos;
short                viewport_id;
gm_$pointreal_t      segment_pos;
status_$t            status;
gm_$event_t          flush_ev_type;
char                 flush_character;
gm_$pointreal_t      flush_bitmap_pos;
short                flush_viewport_id;
gm_$pointreal_t      flush_segment_pos;
status_$t            flush_status;
short                i;
short                j;
short                k;
gm_$point_array16_t  star;
short                vertices;
boolean              closed;
boolean              filled;
gm_$pointreal_t      last_segment_pos;
gm_$point16_t        delta;
short                file_id;
gm_$segment_id_t     sid;
long                 n_instances;
gm_$boundsreal_t     bounds;
boolean              command_picked;
gm_$point16_t        bitmap_size = { 1024, 1024 };
```

```
gm_$pointreal_t      pick_aperture = { 4.0, 4.0 };
gm_$keyset_t         keyset;

/*****************************************************************/
check(status)

status_$t  status;
{
    if(status.all != status_$ok)
        pfm_$error_trap(status);
}
/*****************************************************************/
main()
{
/* Initialize the 2D GMR package. */
    gm_$init( gm_$direct,
              (short)1,
              bitmap_size,
              (short)8,
              status);

    check(status);

/* Create a file named "stars.gmr". */
    gm_$file_create( "stars.gmr",
                     (short)strlen("stars.gmr"),
                     gm_$overwrite,
                     gm_$1w,
                     file_id,
                     status);

    check(status);


/* Coerce REAL data to INTEGER32. */
    gm_$data_coerce_set_real( gm_$32,
                              status);

    check(status);

/* Create an unnamed segment. */
    gm_$segment_create( (char *)NULL,
                        (short)0,
                        sid,
                        status);

    check(status);

/* Define a filled polyline. */
    star[0].x = 000;
    star[0].y = 000;
    star[1].x = 400;
    star[1].y = 300;
    star[2].x = 000;
    star[2].y = 300;
    star[3].x = 400;
    star[3].y = 000;
    star[4].x = 200;
    star[4].y = 400;
    closed = true;
    filled = true;
    vertices = 5;

/* Set model command mode to INSERT. */
    gm_$modelcmd_set_mode( gm_$modelcmd_insert,
                           status);

    check(status);

/* Insert polylines into the segment. */
    for(i=0; i<stars_x; i++)
    {
        for(j=0; j<stars_y; j++)
        {
            gm_$polyline_2d16( vertices,
                               star,
                               closed,
                               filled,
                               status);

            check(status);

            for(k=0; k < vertices; k++)
                star[k].y = star[k].y + 600;
        }/* end for j */
        for(k =0; k < vertices; k++)
        {
            star[k].x = star[k].x + 600;
            star[k].y = star[k].y - 600 * stars_y;
        }/* end for k */
    }/* end for i */

/* Display the segment. */
    gm_$display_segment( sid,
                         status);

    check(status);

/* Set the refresh state to partial. */
    gm_$viewport_set_refresh_state( gm_$refresh_partial,
                                    status);

    check(status);

/* Make the cursor active. */
    gm_$cursor_set_active( true,
                           status);

    check(status);

/* Enable keys ^X, P, and Q. */

    lib_$init_set(keyset, SETSIZE);

    lib_$add_to_set(keyset, SETSIZE, ctrlx);
    lib_$add_to_set(keyset, SETSIZE, 'P');
    lib_$add_to_set(keyset, SETSIZE, 'Q');
    lib_$add_to_set(keyset, SETSIZE, 'p');
    lib_$add_to_set(keyset, SETSIZE, 'q');

    gm_$input_enable( gm_$keystroke,
                      keyset,
                      status);

    check(status);

/* Enable locator events. */

    gm_$input_enable( gm_$locator,
                      OL,
                      status);

    check(status);

    command_picked = false;

    do
    {
/* Wait for an event. */
        gm_$input_event_wait( true,
                              ev_type,
                              character,
                              bitmap_pos,
                              viewport_id,
                              segment_pos,
                              status);

        check(status);

        if(ev_type == gm_$locator)
            do
            {
/* Flush the queue. */

                gm_$input_event_wait( false,
                                      flush_ev_type,
                                      flush_character,
                                      flush_bitmap_pos,
                                      flush_viewport_id,
                                      flush_segment_pos,
                                      flush_status);

                if(flush_ev_type != gm_$no_event)
                {
                    ev_type     = flush_ev_type;
                    character   = flush_character;
                    bitmap_pos  = flush_bitmap_pos;
                    viewport_id = flush_viewport_id;
                    segment_pos = flush_segment_pos;
                    status      = flush_status;
                    check(status);
                }/* end if */
            }while(flush_ev_type == gm_$locator);/* end do */

/* Do case event type. */
            switch (ev_type)
            {
```

```
            case gm_$keystroke:

                switch (character)
                {

                case 'P':
                case 'p':
                    if(! command_picked)
                    {

/* Set the pick center. */
                        gm_$pick_set_center( segment_pos,
                                             status);

                        check(status);

/* Set the pick aperture. */
                        gm_$pick_set_size( pick_aperture,
                                           status);

                        check(status);

/* Clear old pick list. */
                        gm_$pick_segment( gm_$clear,
                                          sid,
                                          n_instances,
                                          bounds,
                                          status);
                        check(status);

/* Setup pick at top segment. */
                        gm_$pick_segment( gm_$setup,
                                          sid,
                                          n_instances,
                                          bounds,
                                          status);
                        if( status.all == status_$ok)
                        {

/* Initialize pick_command. */
                            gm_$pick_command( gm_$start,
                                              status);
                            check(status);

/* Pick a command. */
                            gm_$pick_command( gm_$cnext,
                                              status);

                            command_picked = (status.all == status_$ok);
                        }/* end if */

                        if(command_picked)
                        {

/* Highlight the picked command. */
                            gm_$pick_highlight_command( gm_$outline,
                                                        (float)1.0,
                                                        status);
```

```
                              check(status);

/* Inquire about the picked polyline. */

                              gm_$inq_polyline_2d16( vertices,
                                                     star,
                                                     closed,
                                                     filled,
                                                     status);
                              check(status);

/* Change to rubberband mode. */

                              gm_$modelcmd_set_mode( gm_$modelcmd_rubberband,
                                                     status);

                              check(status);

/* Turn off the cursor. */
                              gm_$cursor_set_active( false,
                                                     status);

                              check(status);
                              last_segment_pos = segment_pos;
                          }/* end if */
                      }/* end if (!command_picked) */
                      else
                      {

/* Change to replace mode. */

                          gm_$modelcmd_set_mode( gm_$modelcmd_replace,
                                                 status);

                          check(status);

/* Replace the polyline. */

                          gm_$polyline_2d16( vertices,
                                             star,
                                             closed,
                                             filled,
                                             status);

                          check(status);

/* Turn the cursor on. */

                          gm_$cursor_set_active( true,
                                                 status);

                          check(status);
                          command_picked = false;
                      }/* end else */
                      break;

                  case ctrlx:

                      command_picked = false;
```

```
/* Turn off rubberband mode. */

                    gm_$modelcmd_set_mode( gm_$modelcmd_replace,
                                           status);

                    check(status);

/* Turn the cursor on. */

                    gm_$cursor_set_active( true,
                                           status);

                    check(status);
                    break;

                case 'Q':
                case 'q':

/* Quit. */             close();
                    exit(0);
                    break;
                }/* end switch character */

            case gm_$locator:

                if(command_picked)
                {/*******************************************************/
 delta.x = (short)floor((double)( segment_pos.x - last_segment_pos.x + 0.5 ));
 delta.y = (short)floor((double)( segment_pos.y - last_segment_pos.y  + 0.5));
 last_segment_pos = segment_pos;
                    for(i=0; i < vertices; i++)
                    {
                        star[i ].x = star[i].x + delta.x;
                    star[i].y = star[i].y + delta.y;
                    }/* end for */

/* Move XOR-rubberband. */
                    gm_$polyline_2d16 ( vertices,
                                        star,
                                        closed,
                                        filled,
                                        status);

                    check(status);
                }/* end if */

                else
                {
                    gm_$cursor_set_position( bitmap_pos,
                                             status );
                    check(status);
                }/* end else */
                break;
        }/* end switch */
    }while(true);/* end do */
}/* end main() */

close()
{
```

```
/* Close the segment. */
    gm_$segment_close( true,
                       status);

    check(status);

/* Close the file. */
    gm_$file_close( true,
                    status);

    check(status);

/* Terminate the session. */

    gm_$terminate(status);

    check(status);
}/* end close */
```

A Program with Attribute Classes and Blocks

The following program creates a hierarchy of segments including instance commands. It displays the file in three viewports; adds attribute class commands to the file; assigns attribute blocks to attribute classes; displays the segments; closes the file; and terminates the package (see Section 13.11).

```
/* attributes */
/* PROGRAM course5 */

#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/time.ins.c"
#include "/sys/ins/gmr.ins.c"
#list

#define aclassid1     1
#define aclassid2     2
#define vpid1         1
#define one_second    250000

gm_$point16_t       bitmap_size = {1024,1024};
status_$t           st;
gm_$boundsreal_t    b;
short               vpid2,vpid3;
short               file_id;
gm_$segment_id_t    sid1,sid2,sid3;
short               ablockid1,ablockid2,ablockid3;
short               ablockid4,ablockid5,ablockid6;
gm_$point16_t       pt1,pt2;
short               lint;
gm_$draw_pattern_t  pattern; /* bit pattern */
time_$clock_t       pause;


main()
{
    gm_$init( gm_$borrow,
              (short)1,
              bitmap_size,
              (short)8,
              st);

/* Create three viewports. */
    b.xmin = 0.0;
    b.ymin = 0.0;
    b.xmax = 0.49;
    b.ymax = 0.49;

    gm_$viewport_set_bounds( b,
                             st);

    b.xmin = 0.51;
    b.ymin = 0.0;
    b.xmax = 1.0;
    b.ymax = 0.49;
    gm_$viewport_create( b,
                         vpid2,
                         st);

    b.xmin = 0.0;
    b.ymin = 0.51;
    b.xmax = 1.0;
    b.ymax = 1.0;

    gm_$viewport_create( b,
                         vpid3,
                         st);

/* Display segments. */

    gm_$file_create("gmfile",
                    (short)6,
                    gm_$overwrite,
                    gm_$1w,
                    file_id,
                    st);

    gm_$segment_create("bottom",
                       (short)6,
                       sid1,
                       st);

    pt1.x = 0;
    pt1.y = 30;
    pt2.x = 10;
    pt2.y = 40;

    gm_$rectangle_16( pt1,
                      pt2,
                      false,
                      st);

    gm_$draw_style( gm_$solid,
                    (short)0,
                    pattern,
                    (short)0,
                    st);
    pt1.x = 20;
    pt2.x = 30;

    gm_$rectangle_16( pt1,
                      pt2,
                      false,
                      st);

    gm_$segment_close( true,
                       st);

    gm_$segment_create("top",
                       (short)3,
                       sid2,
                       st);

    pt1.x = 0;
    pt1.y = 0;
    pt2.x = 10;
    pt2.y = 10;

    gm_$rectangle_16( pt1,
                      pt2,
                      false,
                      st);

    gm_$instance_translate_2d16( sid1,
                                 pt1,
                                 st);
    pt1.x = 20;
    pt2.x = 30;

    gm_$rectangle_16( pt1,
                      pt2,
                      false,
                      st);

    gm_$segment_close( true,
                       st);

    gm_$display_file(st);

/* Display segments in the other two viewports. */

    gm_$viewport_select( vpid2,
                         st);

    gm_$display_file(st);

    gm_$viewport_select( vpid1,
                         st);
    gm_$display_file(st);

    pause.low32 = 5 * one_second;
    pause.high16 = 0;
    time_$wait(time_$relative,pause,st);

/* Assign different attributes to each viewport. */

    gm_$ablock_create( (short)1,
                       ablockid1,
                       st);

    gm_$ablock_set_draw_style( ablockid1,
                               gm_$dotted,
                               (short)5,
                               pattern,
                               (short)0,
                               st);

    gm_$ablock_assign_viewport( aclassid1,
                                vpid1,
                                ablockid1,
                                st);

    gm_$ablock_create( (short)1,
                        ablockid2,
                        st);

    gm_$ablock_set_draw_style( ablockid2,
                               gm_$dotted,
                               (short)10,
                               pattern,
                               (short)0,
                               st);

    gm_$ablock_assign_viewport( aclassid1,
                                vpid2,
                                ablockid2,
                                st);

    gm_$ablock_create( (short)1,
                       ablockid3,
                       st);

    gm_$ablock_set_draw_style( ablockid3,
                               gm_$dotted,
                               (short)20,
                               pattern,
                 .             0,
                               st);

    gm_$ablock_assign_viewport( aclassid1,
                                vpid3,
                                ablockid3,
                                st);

    gm_$display_refresh(st);

    time_$wait( time_$relative,
                pause,
                st);

/* Add an attribute class command. */

    gm_$segment_create("new",
                       (short)3,
                       sid3,
                       st);

    pt1.x = 0;
    pt1.y = 0;
    pt2.x = 10;
    pt2.y = 10;

    gm_$rectangle_16( pt1,
                      pt2,
                      false,
                      st);

    gm_$aclass( aclassid2,
                st);
    pt1.x = 20;
    pt2.x = 30;
    gm_$rectangle_i6( pt1,
                      pt2,
                      false,
                      st);

    gm_$segment_close( true,
                       st);

/* Assign attribute blocks to the attribute class. */

    gm_$ablock_create( (short)1,
                       ablockid4,
                       st);

    gm_$ablock_set_draw_style( ablockid4,
                               gm_$dotted,
                               (short)30,
                               pattern,
                               (short)0,
                               st);

    gm_$ablock_assign_viewport( aclassid2,
                                vpid1,
                                ablockid4,
                                st);

    gm_$ablock_create( (short)1,
                       ablockid5,
                       st);

    gm_$ablock_set_draw_style( ablockid5,
                               gm_$dotted,
                               (short)40,
                               pattern,
                               (short)0,
                               st);

    gm_$ablock_assign_viewport( aclassid2,
                                vpid2,
                                ablockid5,
                                st);

    gm_$ablock_create( (short)1,
                       ablockid6,
                       st);

    gm_$ablock_set_draw_style( ablockid6,
                               gm_$dotted,
                               (short)50,
                               pattern,
                               (short)0,
                               st);

    gm_$ablock_assign_viewport( aclassid2,
                                vpid3,
                                ablockid6,
                                st);

    gm_$display_segment( sid3,
                          st);

    gm_$viewport_select( vpid2,
                          st);

    gm_$display_segment( sid3,
                          st);

    gm_$viewport_select( vpid3,
                          st);

    gm_$display_segment( sid3,
                          st);

    time_$wait( time_$relative,
                pause,
                st);

    gm_$file_close( true,
                    st);

    gm_$terminate(st);
}
```

Program with Advanced Viewing Techniques

The following program changes the color map values; assigns a plane mask to viewports; displays a grid; changes the plane mask; assigns viewport background values; displays segments in more than one viewport; closes the file; and terminates the package (see Section 14.3).

```
/* PROGRAM course6 */

#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/time.ins.c"
#include "/sys/ins/gmr.ins.c"
#include "/sys/ins/pfm.ins.c"
#list


#define gm_default_aclass   1
#define repeats             10
#define space               25
#define one_second          250000

#define plane0      1
#define plane1      2
#define plane2      4


gm_$point16_t      bitmap_size;
status_$t          st;
gm_$boundsreal_t   b;
short              vpid2, vpid3, vpid4;
short              ablockid;
short              k;
short              m ,n;
gm_$color_entry_t  color_array[8];
short              file_id;
gm_$segment_id_t   sid1, sid2, sid3 , sid4;
gm_$point16_t      pt1, pt2, transl;
gm_$pointreal_t    rtransl;
gm_$plane_mask_t   mask;
/***************************************************************/
check(status)

status_$t  status;
{
    if(status.all != status_$ok)
        pfm_$error_trap(status);
}
/***************************************************************/
wait()
{
time_$clock_t  pause;
status_$t      status;

    pause.low32 = 5 * one_second;
    pause.high16 = 0;
```

```
/* Wait five seconds. */
    time_$wait( time_$relative,
                pause,
                status);
    check(status);
}
/*****************************************************************/
main()
{

    bitmap_size.x = 1024;
    bitmap_size.y = 1024;

/* Initialize the 2D GMR package. */

    gm_$init( gm_$borrow,
              stream_$stdout,
              bitmap_size,
              (short)8,
              st);

/* Create and name a metafile. */

    gm_$file_create( "gmfile",
                     (short)6,
                     gm_$overwrite,
                     gm_$1w,
                     file_id,
                      st);

/* Create segment 'grid.' */

    gm_$segment_create( "grid",
                        (short)4,
                        sid1,
                        st);

 /* 'Grid' points are zero-sized rectangles. */

    pt1.x = 0;
    for(m=1; m<=8; m++)
    {
        pt1.x = pt1.x + 100;
        pt1.y = 0;
        for(n=1; n<=8; n++)
        {
            pt1.y = pt1.y + 100;

            gm_$rectangle_16( pt1,
                              pt1,
                              false,
                              st);
        }/* end for n */
    }/* end for m */

 /* Close segment 'grid.' */

    gm_$segment_close( true,
                       st);
    b.xmin = 0.0;
    b.ymin = 0.0;
    b.xmax = 0.49;
    b.ymax = 0.49;

/* Shrink viewport 1. */

    gm_$viewport_set_bounds( b,
                             st);

    b.xmin = 0.51;
    b.ymin = 0.0;
    b.xmax = 1.0;
    b.ymax = 0.49;

/* Create viewport 2. */

    gm_$viewport_create( b,
                         vpid2,
                         st);

    b.xmin = 0.0;
    b.ymin = 0.51;
    b.xmax = 0.49;
    b.ymax = 1.0;

/* Create viewport 3. */

    gm_$viewport_create( b,
                         vpid3,
                         st);

    b.xmin = 0.51;
    b.ymin = 0.51;
    b.xmax = 1.0;
    b.ymax = 1.0;

/* Create viewport 4. */

    gm_$viewport_create( b,
                         vpid4,
                         st);


/* Red + green = yellow. */
    for(k=0; k<8; k++)
    {
        color_array[k].red = 1.0;
        color_array[k].green = 1.0;
        color_array[k].blue  = 0.0;
    }

/* Set color values 8 to 15 to yellow. */

    gm_$display_set_color_map( 0L,
                               0,
                               color_array,
                               st);
```

```
/* Create an ablock. */

    gm_$ablock_create( (short)1,
                       ablockid,
                       st);

/* For the ablock, set the draw value to 9. */

    gm_$ablock_set_draw_value( ablockid,
                               (short)9,
                               st);

/* Assign the ablock to the default aclass. */

    gm_$ablock_assign_display( gm_default_aclass,
                               ablockid,
                               st);

/* Display 'grid' in viewport 4. */

    gm_$display_file(st);

/* Wait a moment. */

    wait();

/* Reset the ablock to default attributes. */

    gm_$ablock_copy( (short)1,
                     ablockid,
                     st);

/* For the ablock, set plane mask to [0,1,2]. */

    mask = ( plane0 | plane1 | plane2 );

    gm_$ablock_set_plane_mask( ablockid,
                               true,
                               mask,
                               st);

/* Create segment 'box.' */

    gm_$segment_create( "box",
                        (short)3,
                        sid2,
                        st);

    pt1.x = 0;
    pt1.y = 0;
    pt2.x = 10;
    pt2.y = 10;

/* Add a rectangle to 'box.' */

    gm_$rectangle_16( pt1,
                      pt2,
                      false,
                      st);
```

```
/* Close segment 'box.' */

    gm_$segment_close( true,
                       st);

/* Create segment 'row.' */

    gm_$segment_create( "row",
                        (short)3,
                        sid3,
                        st);

/* Instance segment 'box' into segment 'row.' */
    transl.y = 0;
    transl.x = 0;
    for(k=0; k<repeats; k++)
    {
        transl.x = transl.x + space;
        gm_$instance_translate_2d16( sid2,
                                     transl,
                                     st);
    }

/* Close segment 'row.' */

    gm_$segment_close( true,
                       st);

 /* Create segment 'block.' */

    gm_$segment_create( "block",
                        5,
                        sid4,
                        st);

/* Instance segment 'row' into segment 'block.' */

    transl.y = 50;
    for(k=0; k<repeats; k++)
    {
        transl.x = k ;
        transl.y = transl.y - space;
        gm_$instance_translate_2d16( sid3,
                                     transl,
                                     st);
    }

/* Close segment 'block.' */

    gm_$segment_close( true, st);

/* Display segment 'block' in viewport 3. */

    gm_$display_segment( sid4,
                         st);

/* Wait a moment. */
    wait();
    rtransl.x = 0.5;
    rtransl.y = 1.0;

/* For viewport 3, zoom out. */

    gm_$view_scale( (float)0.25,
                    rtransl,
                    st);

    rtransl.x = -0.06;
    rtransl.y = 0.0;

/* For viewport 3, pan from left to right. */
    for(k=0; k<5; k++)
        gm_$view_translate( rtransl,
                            st);

    rtransl.x = 0.5;
    rtransl.y = 0.5;

/* For viewport 3, pan diagonally towards lower left. */
    for(k=0; k<5; k++)
        gm_$view_scale( (float)0.85,
                        rtransl,
                        st);

/* Wait a moment. */
    wait();

/* For viewport 2, set the background value to 2. */

    gm_$viewport_set_background_value( vpid2,
                                       2L,
                                       st);

/* Select viewport 2. */

    gm_$viewport_select( vpid2,
                         st);

/* Display segment 'row' in viewport 2. */

    gm_$display_segment( sid3,
                         st);

/* Wait a moment. */
    wait();

/* For viewport 3, set background value to 2. */

    gm_$viewport_set_background_value( vpid3,
                                       3L,
                                       st);

/* Select viewport 3. */

    gm_$viewport_select( vpid3,
                         st);
```

```
/* Display segment 'block' in viewport 3. */

    gm_$display_segment( sid4,
                         st);

/* Wait a moment. */
    wait();

/* Close the file. */
    gm_$file_close( true,
                    st);


/* Terminate the 2D GMR package. */
    gm_$terminate(st);
}
```

Attributes and Instancing

The following program displays a file as it is being created and edited. The file creates the picture in Figure D-1.

```
/* PROGRAM hotel.pas */


#nolist
#include <stdio.h>
#include "/sys/ins/base.ins.c"
#include "/sys/ins/error.ins.c"
#include "/sys/ins/pfm.ins.c"
#include "/sys/ins/gmr.ins.c"
#include "/sys/ins/time.ins.c"
#list


#define one_second      250000
#define five_seconds    (5 * one_second)

status_$t            status;
short                file_id;
short                font_file_id;
gm_$segment_id_t     sid_scene;
gm_$segment_id_t     sid_door;
gm_$segment_id_t     sid_window;
gm_$segment_id_t     sid_sign;
gm_$segment_id_t     sid_tree;
gm_$segment_id_t     sid_house;
gm_$draw_pattern_t   pattern;
gm_$point_array16_t  p;
gm_$point16_t        center;
short                radius;
short                i;
time_$clock_t        pause;
/***************************************************************/
check(status)

status_$t  status;
{
    if(status.all != status_$ok)
        pfm_$error_trap(status);
}
/***************************************************************/
main()
{
/* Intialize GMR package. */
    p[0].x = 1024;
    p[0].y = 1024;

    gm_$init( gm_$direct,
              stream_$stdout,
              p[0],
              (short)8,
              status);

    check(status);
```

```
/* Create and name a metafile. */

    gm_$file_create( "hotel.gm",
                     (short)8,
                      gm_$overwrite,
                      gm_$1w,
                      file_id,
                      status);
    check(status);

/* Load a font family. */

    gm_$font_family_include( "ff0",
                             (short)3,
                              gm_$pixel,
                              font_file_id,
                              status);
    check(status);

/* Set the data coerce. */

    gm_$data_coerce_set_real( gm_$32,
                              status);
    check(status);


/* Create the segment for the door. */

    gm_$segment_create( (char *)NULL,
                         (short)0,
                         sid_door,
                         status);
    check(status);

/* Construct the door. */
    p[0].x = 0;
    p[0].y = 0;
    p[1].x = 36;
    p[1].y = 80;

    gm_$rectangle_16( p[0],
                      p[1],
                      true,
                      status);
    check(status);

/* Construct the door knob. */

    gm_$fill_value( 0L,
                    status);
    check(status);

    p[0].x = 30;
    p[0].y = 38;
    p[1].x = 33;
    p[1].y = 41;

    gm_$rectangle_16( p[0],
                      p[1],
                    true,
                    status);
    check(status);

    gm_$segment_close( true,
                       status);
    check(status);

/* Create the segment for the windows. */
    gm_$segment_create( (char *)NULL,
                        (short)0,
                        sid_window,
                        status);
    check(status);

    p[0].x = 0;
    p[0].y = 0;
    p[1].x = 36;
    p[1].y = 36;

    gm_$rectangle_16( p[0],
                      p[1],
                      false,
                      status);

    check(status);

    p[0].x = 0;
    p[0].y = 18;
    p[1].x = 36;
    p[1].y = 18;

    gm_$polyline_2d16( (short)2,
                        p,
                        false,
                        false,
                        status);
    check(status);

    p[0].x = 18;
    p[0].y = 0;
    p[1].x = 18;
    p[1].y = 36;

    gm_$polyline_2d16( (short)2,
                        p,
                        false,
                        false,
                        status);
    check(status);

    gm_$segment_close( true,
                       status);
    check(status);


/* Create the segment for the sign. */
    gm_$segment_create( (char *)NULL,
                        (short)0,
                          sid_sign,
                          status);

    check(status);

    gm_$text_size( (float)14.0,
                   status);

    check(status);

    p[0].x = 0;
    p[0].y = 0;

    gm_$text_2d16( p[0],
                   (float)0.0,
                   "GRAND MOTEL",
                   (short)11,
                   status);

    check(status);

    gm_$segment_close( true,
                       status);

    check(status);


/* Create the segment for the house. */
    gm_$segment_create( (char *)NULL,
                        (short)0,
                        sid_house,
                        status);

    check(status);

 /* Build the house. */
    p[0].x = 0;
    p[0].y = 0;
    p[1].x = 480;
    p[1].y = 260;

    gm_$rectangle_16( p[0],
                      p[1],
                      false,
                      status);

    check(status);

/* Build the roof. */
    p[0].x = -10;
    p[0].y = 255;
    p[1].x = 240;
    p[1].y = 380;
    p[2].x = 490;
    p[2].y = 255;

    gm_$polyline_2d16( (short)3,
                       p,
                       false,
                     false,
                     status);

    check(status);

/* Build the chimney. */
    p[0].x = 300;
    p[0].y = 350;
    p[1].x = 300;
    p[1].y = 370;
    p[2].x = 330;
    p[2].y = 370;
    p[3].x = 330;
    p[3].y = 335;

    gm_$polyline_2d16( (short)4,
                        p,
                        false,
                        false,
                        status);

    check(status);

/* Build the round window. */
    center.x = 240;
    center.y = 195;
    radius = 45;

    gm_$circle_16( center,
                   radius,
                   false,
                   status);

    check(status);

    p[0].x = center.x - radius;
    p[0].y = center.y;
    p[1].x = center.x + radius;
    p[1].y = center.y;
    p[2].x = center.x;
    p[2].y = center.y - radius;
    p[3].x = center.x;
    p[3].y = center.y + radius;

    gm_$polyline_2d16( (short)4,
                       p,
                       true,
                       false,
                       status);

    check(status);

    p[4].x = p[1].x;
    p[4].y = p[1].y;
    p[1].x = p[2].x;
    p[1].y = p[2].y;

    gm_$polyline_2d16( (short)2,
                       p,
                        false,
                        false,
                        status);

    check(status);

    gm_$polyline_2d16( (short)2,
                           p[3],
                           false,
                           false,
                           status);

    check(status);

/* Instance and position the door. */
    p[0].x = 222;
    p[0].y = 0;

    gm_$instance_translate_2d16( sid_door,
                                 p[0],
                                 status);
    check(status);

/* Instance and position the windows. */
    p[0].x = 50;
    p[0].y = 40;

    gm_$instance_translate_2d16( sid_window,
                                 p[0],
                                 status);

    check(status);

    p[0].x = 118;
    gm_$instance_translate_2d16( sid_window,
                                 p[0],
                                 status);

    check(status);

    p[0].x = 326;
    gm_$instance_translate_2d16( sid_window,
                                 p[0],
                                 status);

    check(status);

    p[0].x = 394;

    gm_$instance_translate_2d16( sid_window,
                                 p[0],
                                 status);

    check(status);

    p[0].y = 180;

    gm_$instance_translate_2d16( sid_window,
                                 p[0],
                                  status);

    check(status);

    p[0].x = 326;

    gm_$instance_translate_2d16( sid_window,
                                 p[0],
                                 status);
    check(status);

    p[0].x = 118;

    gm_$instance_translate_2d16( sid_window,
                                 p[0],
                                 status);

    check(status);

    p[0].x = 50;

    gm_$instance_translate_2d16( sid_window,
                                 p[0],
                                 status);

    check(status);


/* Instance and position the segment for the sign. */
    p[0].x = 172;
    p[0].y = 120;

    gm_$instance_translate_2d16( sid_sign,
                                 p[0],
                                 status);

    check(status);

    gm_$segment_close( true,
                       status);

    check(status);


/* Create the segment for the trees. */
    gm_$segment_create( (char *)NULL,
                        (short)0,
                        sid_tree,
                        status);

    check(status);

    p[0].x = 0;
    p[0].y = 0;
    p[1].x = 0;
    p[1].y = 150;
    gm_$polyline_2d16( (short)2,
                       p,
                       false,
                    false,
                    status);

check(status);

p[0].x = 12;
p[1].x = 12;

gm_$polyline_2d16( (short)2,
                    p,
                    false,
                    false,
                    status);

check(status);

p[0].x = 6;
p[0].y = 200;
gm_$circle_16( p[0],
               (short)50,
               false,
               status);

check(status);

gm_$draw_style( gm_$dotted,
                (short)2,
                pattern,
                (short)0,
                status);

p[0].x = 0;
p[0].y = 180;
p[1].x = -40;
p[1].y = 200;

gm_$polyline_2d16( (short)2,
                   p,
                   false,
                   false,
                   status);

check(status);

p[0].x = 12;
p[1].x = 52;

gm_$polyline_2d16( (short)2,
                    p,
                    false,
                    false,
                    status);

check(status);

p[0].x = 4;
p[0].y = 190;
p[1].x = -20;
p[1].y = 230;
```

```
gm_$polyline_2d16( (short)2,
                    p,
                    false,
                    false,
                    status);

check(status);

p[0].x = 8;
p[0].y = 190;
p[1].x = 32;
p[1].y = 230;

gm_$polyline_2d16( (short)2,
                   p,
                   false,
                   false,
                   status);

check(status);

p[0].x = 6;
p[0].y = 195;
p[1].x = 6;
p[1].y = 240;

gm_$polyline_2d16( (short)2,
                    p,
                    false,
                    false,
                    status);
check(status);

p[0].x = 0;
p[0].y = 170;
p[1].x = 0;
p[1].y = 150;

gm_$polyline_2d16( (short)2,
                    p,
                    false,
                    false,
                    status);

check(status);

p[0].x = 12;
p[1].x = 12;

gm_$polyline_2d16( (short)2,
                   p,
                   false,
                   false,
                   status);

check(status);

gm_$segment_close( true,
                   status);
```

```
    check(status);


/* Create the segment called scene. */

    gm_$segment_create( (char *)NULL,
                        (short)0,
                         sid_scene,
                         status);

    check(status);

    p[0].x = 0;                          /* Instance the segment HOUSE. */
    p[0].y = 0;

    gm_$instance_translate_2d16( sid_house,
                                 p[0],
                                 status);

    check(status);

/* Instance, translate, and scale the segment the trees. */
    p[0].x = -85;
    p[0].y = -25;

    gm_$instance_scale_2d16( sid_tree,
                             (float)2.0,
                              p[0],
                              status);

    p[0].x = 530;
    p[0].y = 55;

    gm_$instance_scale_2d16( sid_tree,
                             (float)0.75,
                             p[0],
                             status);

    check(status);

    p[0].x = 610;
    p[0].y = 105;

    gm_$instance_scale_2d16( sid_tree,
                             (float)0.85,
                             p[0],
                             status);

    check(status);

    gm_$segment_close( true,
                       status);

    check(status);


/* Now display the completed scene. */

gm_$display_segment( sid_scene,
                    status);

    check(status);

/* Admire the scene for five seconds. */
    pause.low32  = five_seconds;
    pause.high16 = 0;

    time_$wait( time_$relative,
                pause,
                status);

    check(status);


/* Close and save the file. */

    gm_$file_close( true,
                    status);

    check(status);

/* Terminate the 2D GMR package. */
    gm_$terminate(status);

    check(status);
}
```

Appendix F
FORTRAN Program Examples

This Appendix contains some of the programming examples presented in the manual translated into FORTRAN.

A Program to Draw a Rectrangle

The following program demonstrates how to initialize the 2D GMR package, create a metafile, create a segment, and draw a rectangle (see Section 3.8 and Figure 3-3).

```
       program draw_rectangle
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gmr.ins.ftn'
%include '/sys/ins/time.ins.ftn'

      integer*2          file_id
      integer*4          segment_id
      integer*4          st
      integer*2          pt1(2),pt2(2)
      integer*4          i
      integer*2          bitmap_size(2)
      integer*2          pause(3)
      integer*2          high_plane

c                    { Define the coordinates of the rectangle to be drawn. }
      pt1(1) = 100
      pt1(2) = 30
      pt2(1) = 200
      pt2(2) = 50

      bitmap_size(1) = 1024
      bitmap_size(2) = 1024

c                                                  { Initialize 2D GMR. }
      call gm_$init(
     +        gm_$direct
     +        ,stream_$stdout
     +        ,bitmap_size
     +        ,high_plane
     +        ,st
     +        )

c                                            { Create and name metafile. }
      call gm_$file_create(
     +        'gmfile'
     +        ,int2(6)
     +        ,gm_$overwrite
     +        ,gm_$1w
     +        ,file_id
     +        ,st
     +        )

c                                            { Create and name a segment.}
```

```
      call gm_$segment_create
     +        ('rectang_seg'
     +        ,int2(11)
     +        ,segment_id
     +        ,st
     +        )

c                                                         { Insert a rectangle. }
      call gm_$rectangle_16
     +        (pt1
     +        ,pt2
     +        ,false
     +        ,st
     +        )

c                                                           { Display the file. }
      call gm_$display_file
     +        (
     +        st
     +        )

c                                           { Display the figure for five seconds. }
      pause(1) = 0
      pause(2) = 20
      pause(3) = 0
      call time_$wait(
     +        time_$relative
     +        ,pause
     +        ,st
     +        )

c                                                          { Close the segment. }
      call gm_$segment_close(
     +        true
     +        ,st
     +        )

c                                                         { Close the metafile. }
      call gm_$file_close(
     +        true
     +        ,st
     +        )

      call gm_$terminate(
     +        st
     +        )

      END
```

A Program Using Primitives and Instancing

The following program draws the design in Figure 4-2 using primitives and instancing (see Section 4.6).

```
      program draw_rectangle

%nolist
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gmr.ins.ftn'
%include '/sys/ins/time.ins.ftn'
%list


      integer*2         file_id
      integer*4         small_id
      integer*4         large_id
      integer*4         st
      integer*2         pt1(2), pt2(2)
      integer*4         i
      integer*2         bitmap_size(2)
      integer*2         position(2)
      integer*2         positions(4)
      character         pattern(8)
      integer*2         pause(3)


      bitmap_size(1) = 1024
      bitmap_size(2) = 1024

c                                                       { Initialize 2D GMR. }
      call gm_$init
     +         (gm_$direct
     +         ,stream_$stdout
     +         ,bitmap_size
     +         ,int2(8)
     +         ,st
     +         )
      call error(st)

c                                               { Create and name a metafile. }
      call gm_$file_create
     +         ('gmfile'
     +         ,int2(6)
     +         ,gm_$overwrite
     +         ,gm_$1w
     +         ,file_id
     +         ,st
     +         )
      call error(st)

c                                       { Define coordinates of a rectangle. }
      pt1(1) = 100
      pt1(2) = 100
      pt2(1) = 200
      pt2(2) = 200
```

```
c                                                { Create and name a segment. }
      call gm_$segment_create
     +        ('small_rec'
     +        ,int2(9)
     +        ,small_id
     +        ,st
     +        )
      call error(st)

c                                                         { Draw a rectangle. }
      call gm_$rectangle_16
     +        (pt1
     +        ,pt2
     +        ,.true.
     +        ,st
     +        )
      call error(st)

c                                                        { Close the segment. }
      call gm_$segment_close
     +        (.true.
     +        ,st
     +        )
      call error(st)



      call gm_$segment_create
     +        ('large_rec'
     +        ,int2(9)
     +        ,large_id
     +        ,st
     +        )
      call error(st)

      call gm_$draw_style
     +        (gm_$solid
     +        ,int2(4)
     +        ,pattern
     +        ,int2(0)
     +        ,st
     +        )
      call error(st)

c                     { Define the coordinates of the rectangle to be drawn. }
      pt1(1) = 100
      pt1(2) = 100
      pt2(1) = 600
      pt2(2) = 600


c                                               { Draw an unfilled rectangle. }
      call gm_$rectangle_16
     +        (pt1
     +        ,pt2
     +        ,.false.
     +        ,st
     +        )
      call error(st)
      position(1) = 100
      position(2) = 100

c                                              { Instance the rectange four times. }
      call gm_$instance_translate_2d16
     +        (small_id
     +        ,position
     +        ,st
     +        )
      call error(st)

      position(1) = 300
      position(2) = 300

      call gm_$instance_translate_2d16
     +        (small_id
     +        ,position
     +        ,st
     +        )
      call error(st)

      position(1) = 300
      position(2) = 100

      call gm_$instance_translate_2d16
     +        (small_id
     +        ,position
     +        ,st
     +        )
      call error(st)

      position(1) = 100
      position(2) = 300

      call gm_$instance_translate_2d16
     +        (small_id
     +        ,position
     +        ,st
     +        )
      call error(st)

      positions(1) = 300
      positions(2) = 300
      positions(3) = 400
      positions(4) = 400

c                                          { Draw a line connecting the rectangles. }
      call gm_$polyline_2d16
     +        (int2(2)
     +        ,positions
     +        ,.false.
     +        ,.false.
     +        ,st
     +        )
      call error(st)

      positions(1) = 300
      positions(2) = 400
      positions(3) = 400
      positions(4) = 300


      call gm_$polyline_2d16
     +        (int2(2)
     +        ,positions
     +        ,.false.
     +        ,.false.
     +        ,st
     +        )
      call error(st)


c                                                  { Close the segment. }
      call gm_$segment_close
     +        (.true.
     +        ,st
     +        )
      call error(st)

      call gm_$display_segment
     +        (large_id
     +        ,st
     +        )
      call error(st)


c                                   { Display the figure for five seconds. }
      pause(1) = 0
      pause(2) = 20
      pause(3) = 0
      call time_$wait
     +        ( time_$relative
     +        , pause
     +        , st
     +        )
      call error(st)

c                                                  { Close the metafile. }
      call gm_$file_close
     +        (.true.
     +        ,st
     +        )
      call error(st)

      call gm_$terminate
     +        (
     +        st
     +        )
      call error(st)

      END
c ****************************************************

      subroutine error(st)
      integer*4 st
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'
      if (st .ne. 0) then
```

```
   call pfm_$error_trap(st)
endif
return
end
```

A Program Including Text

The following program draws the design in Figure 6-1. This is a rectangle with horizontal and vertical text strings (see Section 6.5).

```
      program draw_rectangle_text

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/gmr.ins.ftn'
%include '/sys/ins/time.ins.ftn'
%include '/sys/ins/error.ins.ftn'


      integer*2           file_id
      integer*4           segment_id
      integer*4           st
      integer*2           pt1(2), pt2(2),point(2)
      integer*4           i
      integer*2           bitmap_size(2)
      integer*2           ffid
      integer*2           pause(3)

      bitmap_size(1) = 1024
      bitmap_size(2) = 1024

c                                                  { Initialize 2D GMR. }
      call gm_$init
     +     (gm_$direct
     +     ,stream_$stdout
     +     ,bitmap_size
     +     ,int2(8)
     +     ,st
     +     )
      call error(st)

c                                           { Create and name a metafile. }
      call gm_$file_create
     +     ('gmfile'
     +     ,int2(6)
     +     ,gm_$overwrite
     +     ,gm_$1w
     +     ,file_id
     +     ,st
     +     )
      call error(st)

c                                            { Create and name a segment. }
      call gm_$segment_create
     +   ('rectang_seg',
     +   int2(11)
     +   ,segment_id
     +   ,st
     +   )
      call error(st)

c                                                { Load the font family. }
      call gm_$font_family_include
     +     ( 'font_families'
     +     , int2(13)
     +     , gm_$pixel
     +     , ffid
     +     , st
     +     )
      call error(st)

      call gm_$text_size
     +     ( 14.0
     +     , st
     +     )
      call error(st)

      point(1) = 5
      point(2) = 510

c                                              { Display a line of text. }
      call gm_$text_2d16
     +     ( point, 0.0
     +     , 'This is the top of the rectangle.'
     +     , int2(33)
     +     , st
     +     )
      call error(st)

      point(1) = 5
      point(2) = 50

c                                              { Display a line of text. }
      call gm_$text_2d16
     +     ( point
     +     , -90.0
     +     , 'This is the side of the rectangle.'
     +     , int2(34)
     +     , st
     +     )
      call error(st)

c                    { Define the coordinates of the rectangle to be drawn. }
      pt1(1) = 10
      pt1(2) = 30
      pt2(1) = 400
      pt2(2) = 500

c                                                    { Draw a rectangle. }
      call gm_$rectangle_16
     +     (pt1
     +     ,pt2
     +     ,false
     +     ,st
     +     )
      call error(st)

c                                                   { Close the segment. }
      call gm_$segment_close
     +     (.true.
     +     ,st
     +     )
      call error(st)
```

```
c     {Display the file.}
      call gm_$display_file
     +     (st
     +     )
      call error(st)

c                                    { Display the figure for five seconds. }
      pause(1) = 0
      pause(2) = 20
      pause(3) = 0
      call time_$wait
     +     ( time_$relative
     +     , pause
     +     , st
     +     )
      call error(st)

c                                                    { Close the metafile. }
      call gm_$file_close
     +     ( true
     +     ,st
     +     )
      call error(st)

c                                                      { Terminate 2D GMR. }
      call gm_$terminate
     +     ( st
     +     )
      call error(st)

      END

c **************************************************

      subroutine error(st)
      integer*4 st
%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'
      if (st .ne. 0) then
         call pfm_$error_trap(st)
      endif
      return
      end
```

A Program with Attribute Classes and Blocks

The following program creates a hierarchy of segments including instance commands. It displays the file in three viewports; adds attribute class commands to the file; assigns attribute blocks to attribute classes; displays the segments; closes the file; and terminates the package (see Chapter 13).

```
      PROGRAM course5

%INCLUDE '/sys/ins/base.ins.ftn'
%INCLUDE '/sys/ins/time.ins.ftn'
%INCLUDE '/sys/ins/gmr.ins.ftn'


      integer*2          aclassid1
      integer*2          aclassid2
      integer*2          vpid1
      integer*2          vpid2
      integer*2          vpid3
      integer*2          bitmap_size(2)
      integer*4          status
      real               b(4)
      integer*2          file_id
      integer*4          sid1
      integer*4          sid2
      integer*4          sid3
      integer*2          ablockid1
      integer*2          ablockid2
      integer*2          ablockid3
      integer*2          ablockid4
      integer*2          ablockid5
      integer*2          ablockid6
      integer*2          pt1(2)
      integer*2          pt2(2)
      character          pattern(8)
      integer*2          pause(3)

      aclassid1    = 1       { Default aclass }
      aclassid2    = 2
      vpid1        = 1       { Initial viewport }


      bitmap_size(1) = 1024
      bitmap_size(2) = 1024

c                                     { Initialize the 2D GMR package. }
      call gm_$init
     +       ( gm_$borrow
     +       , stream_$stdout
     +       , bitmap_size
     +       , int2(8)
     +       , status
     +       )

      b(1) = 0.00
      b(2) = 0.00
      b(3) = 0.49
      b(4) = 0.49

c                                                  { Create viewport 1. }
      call gm_$viewport_set_bounds
     +      ( b
     +      , status
     +      )

      b(1) = 0.51
      b(2) = 0.00
      b(3) = 1.00
      b(4) = 0.49

c                                                  { Create viewport 2. }
      call gm_$viewport_create
     +      ( b
     +      , vpid2
     +      , status
     +      )

      b(1) = 0.00
      b(2) = 0.51
      b(3) = 1.00
      b(4) = 1.00

c                                                  { Create viewport 3. }
      call gm_$viewport_create
     +      ( b
     +      , vpid3
     +      , status
     +      )


c                                                      { Create a file. }
      call gm_$file_create
     +      ( 'gmfile'
     +      , int2(6)
     +      , gm_$overwrite
     +      , gm_$1w
     +      , file_id
     +      , status
     +      )

c                                             { Create 'bottom' segment. }
      call gm_$segment_create
     +      ( 'bottom'
     +      , int2(6)
     +      , sid1
     +      , status
     +      )

      pt1(1) = 0
      pt1(2) = 30
      pt2(1) = 10
      pt2(2) = 40

c                                     { Add a rectangle to 'bottom' segment. }
      call gm_$rectangle_16
     +      ( pt1
     +      , pt2
     +      , .FALSE.
     +      , status
     +      )

c                                       { Change the draw style to solid. }
      call gm_$draw_style
     +      ( gm_$solid
     +      , int2(0)
     +      , pattern
     +      , int2(0)
     +      , status
     +      )

      pt1(1) = 20
      pt2(1) = 30

c                                  { Add a rectangle to 'bottom' segment. }
      call gm_$rectangle_16
     +      ( pt1
     +      , pt2
     +      , .FALSE.
     +      , status
     +      )

c                                              { Close 'bottom' segment. }
      call gm_$segment_close
     +      ( .TRUE.
     +      , status
     +      )

c                                                 { Create 'top' segment. }
      call gm_$segment_create
     +      ( 'top'
     +      , int2(3)
     +      , sid2
     +      , status
     +      )

      pt1(1) = 0
      pt1(2) = 0
      pt2(1) = 10
      pt2(2) = 10

c                                     { Add a rectangle to 'top' segment. }
      call gm_$rectangle_16
     +      ( pt1
     +      , pt2
     +      , .FALSE.
     +      , status
     +      )

c                                 { Instance 'bottom' into 'top' segment. }
      call gm_$instance_translate_2d16
     +      ( sid1
     +      , pt1
     +      , status
     +      )
```

```
      pt1(1) = 20
      pt2(1) = 30

c                                  { Add a rectangle to 'top' segment. }
      call gm_$rectangle_16
     +      ( pt1
     +      , pt2
     +      , .FALSE.
     +      , status
     +      )

c                                              { Close 'top' segment. }
      call gm_$segment_close
     +      ( .TRUE.
     +      , status
     +      )

c                                    { Display the file in viewport 3. }
      call gm_$display_file
     +      ( status
     +      )

c                                                { Select viewport 2. }
      call gm_$viewport_select
     +      ( vpid2
     +      , status
     +      )

c                                    { Display the file in viewport 2. }
      call gm_$display_file
     +      ( status
     +      )

c                                                { Select viewport 1. }
      call gm_$viewport_select
     +      ( vpid1
     +      , status
     +      )

c                                        { Display file in viewport 1. }
      call gm_$display_file
     +      ( status
     +      )

      pause(1) = 0
      pause(2) = 20
      pause(3) = 0
      call time_$wait
     +      ( time_$relative
     +      , pause
     +      , status
     +      )

c                                                 { Create ablockid1. }
      call gm_$ablock_create
     +      ( int2(1)
     +      , ablockid1
     +      , status
     +      )
```

```
c                                        { Give ablockid1 the dotted line style }
c                                               {   with repetition factor = 5. }
      call gm_$ablock_set_draw_style
     +      ( ablockid1
     +      , gm_$dotted
     +      , int2(5)
     +      , pattern
     +      , int2(0)
     +      , status
     +      )

c                               { Assign ablockid1 to aclassid1 in viewport 1. }
      call gm_$ablock_assign_viewport
     +      ( aclassid1
     +      , vpid1
     +      , ablockid1
     +      , status
     +      )

c                                                          { Create ablockid2. }
      call gm_$ablock_create
     +      ( int2(1)
     +      , ablockid2
     +      , status
     +      )

c                                        { Give ablockid2 the dotted line style }
c                                              {   with repetition factor = 10. }
      call gm_$ablock_set_draw_style
     +      ( ablockid2
     +      , gm_$dotted
     +      , int2(10)
     +      , pattern
     +      , int2(0)
     +      , status
     +      )

c                               { Assign ablockid1 to aclassid2 in viewport 2. }
      call gm_$ablock_assign_viewport
     +      ( aclassid1
     +      , vpid2
     +      , ablockid2
     +      , status
     +      )

c                                                          { Create ablockid3. }
      call gm_$ablock_create
     +      ( int2(1)
     +      , ablockid3
     +      , status
     +      )

c                                       { Give ablockid3 the dotted line style. }
c                                               {   with repetition factor = 20 }
      call gm_$ablock_set_draw_style
     +      ( ablockid3
     +      , gm_$dotted
     +      , int2(20)
     +      , pattern
     +      , int2(0)
     +      , status
     +      )

c                          { Assign ablockid3 to aclassid1 in viewport 3. }
      call gm_$ablock_assign_viewport
     +      ( aclassid1
     +      , vpid3
     +      , ablockid3
     +      , status
     +      )

c                                  { Refresh display to see the effects of }
c                                                 {   the attribute blocks. }
      call gm_$display_refresh
     +      ( status
     +      )

      call time_$wait
     +      ( time_$relative
     +      , pause
     +      , status
     +      )

c                                                  { Create 'new' segment. }
      call gm_$segment_create
     +      ( 'new'
     +      , int2(3)
     +      , sid3
     +      , status
     +      )

      pt1(1) = 0
      pt1(2) = 0
      pt2(1) = 10
      pt2(2) = 10

c                                      { Add a rectangle to 'new' segment. }
      call gm_$rectangle_16
     +      ( pt1
     +      , pt2
     +      , .FALSE.
     +      , status
     +      )

c                                 { Add an aclass command to 'new' segment. }
      call gm_$aclass
     +      ( aclassid2
     +      , status
     +      )

      pt1(1) = 20
      pt2(1) = 30

c                                      { Add a rectangle to 'new' segment. }
      call gm_$rectangle_16
     +      ( pt1
     +      , pt2
     +      , .FALSE.
     +      , status
     +      )

c                                                 { Close 'new' segment. }
      call gm_$segment_close
     +      ( .TRUE.
     +      , status
     +      )

c                                                    { Create ablockid4. }
      call gm_$ablock_create
     +      ( int2(1)
     +      , ablockid4
     +      , status
     +      )

c                                    { Give ablockid4 the dotted line style }
c                                          {   with repetition factor = 30. }
      call gm_$ablock_set_draw_style
     +      ( ablockid4
     +      , gm_$dotted
     +      , int2(30)
     +      , pattern
     +      , int2(0)
     +      , status
     +      )

c                            { Assign ablockid4 to aclassid2 in viewport 1. }
      call gm_$ablock_assign_viewport
     +      ( aclassid2
     +      , vpid1
     +      , ablockid4
     +      , status
     +      )

c                                                    { Create ablockid5. }
      call gm_$ablock_create
     +      ( int2(1)
     +      , ablockid5
     +      , status
     +      )

c                                    { Give ablockid5 the dotted line style }
c                                          {   with repetition factor = 40. }
      call gm_$ablock_set_draw_style
     +      ( ablockid5
     +      , gm_$dotted
     +      , int2(40)
     +      , pattern
     +      , int2(0)
     +      , status
     +      )

c                            { Assign ablockid5 to aclassid2 in viewport 2. }
      call gm_$ablock_assign_viewport
     +      ( aclassid2
     +      , vpid2
     +      , ablockid5
     +      , status
     +        )

c                                                    { Create ablockid6. }
      call gm_$ablock_create
     +        ( int2(1)
     +        , ablockid6
     +        , status
     +        )

c                                     { Give ablockid6 the dotted line style }
c                                           {   with repetition factor = 50. }
      call gm_$ablock_set_draw_style
     +        ( ablockid6
     +        , gm_$dotted
     +        , int2(50)
     +        , pattern
     +        , int2(0)
     +        , status
     +        )

c                              { Assign ablockid6 to aclassid2 in viewport 3. }
      call gm_$ablock_assign_viewport
     +        ( aclassid2
     +        , vpid3
     +        , ablockid6
     +        , status
     +        )

c                                    { Display 'new' segment in viewport 1. }
      call gm_$display_segment
     +        ( sid3
     +        , status
     +        )

c                                                   { Select viewport 2. }
      call gm_$viewport_select
     +        ( vpid2
     +        , status
     +        )

c                                     { Display 'new' segment in viewport 2. }
      call gm_$display_segment
     +        ( sid3
     +        , status
     +        )

c                                                   { Select viewport 3. }
      call gm_$viewport_select
     +        ( vpid3
     +        , status
     +        )

c                                     { Display 'new' segment in viewport 3. }
      call gm_$display_segment
     +        ( sid3
     +        , status
     +        )

      call time_$wait
      +       ( time_$relative
      +       , pause
      +       , status
      +       )

c                                                { Close the file. }
       call gm_$file_close
      +       ( .TRUE.
      +       , status
      +       )

c                                   { Terminate the 2D GMR package. }
       call gm_$terminate
      +       ( status
      +       )

       END
```

Program with Advanced Viewing Techniques

The following program changes the color map values; assigns a plane mask to viewports; displays a grid; changes the plane mask; assigns viewport background values; displays segments in more than one viewport; closes the file; and terminates the package (see Section 14.3).

```
      PROGRAM course6

%INCLUDE '/sys/ins/base.ins.ftn'
%INCLUDE '/sys/ins/time.ins.ftn'
%INCLUDE '/sys/ins/gmr.ins.ftn'
%INCLUDE '/sys/ins/pfm.ins.ftn'


      integer*2                gm_default_aclass
      integer*2                repeats
      integer*2                space


      integer*2                bitmap_size(2)
      integer*2                st
      real                     b(4)
      integer*2                vpid2
      integer*2                vpid3
      integer*2                vpid4

      integer*2                ablockid
      integer*2                k
      integer*2                m
      integer*2                n
      real                     color_array(8:15)
      integer*2                file_id
      integer*4                sid1
      integer*4                sid2
      integer*4                sid3
      integer*4                sid4
      integer*2                pt1(2)
      integer*2                pt2(2)
      integer*2                transl(2)
      real                     rtransl(2)



      gm_default_aclass = 1
      repeats           = 10
      space             = 25

      bitmap_size(1) = 1024
      bitmap_size(2) = 1024

c                                          { Initialize the 2D GMR package. }
      call gm_$init
     +      ( gm_$borrow
     +      , stream_$stdout
     +      , bitmap_size
     +      , int2(8)
     +      , st
     +      )

c                                                 { Create and name a metafile. }
      call gm_$file_create
     +      ( 'gmfile'
     +      , int2(6)
     +      , gm_$overwrite
     +      , gm_$1w
     +      , file_id
     +      , st
     +      )

c                                                      { Create segment 'grid.' }
      call gm_$segment_create
     +      ( 'grid'
     +      , int2(4)
     +      , sid1
     +      , st
     +      )

c                                   { 'Grid' points are zero-sized rectangles. }
      pt1(1) = 0
      Do 100 m = 1,8
            pt1(1) = pt1(1) + 100
            pt1(2) = 0
            Do 50  n = 1,8
                  pt1(2) = pt1(2) + 100
                  call gm_$rectangle_16
     +                  ( pt1
     +                  , pt1
     +                  , .FALSE.
     +                  , st
     +                  )
50          continue
100   continue

c                                                      { Close segment 'grid.' }
      call gm_$segment_close
     +      ( .TRUE.
     +      , st
     +      )

      b(1) = 0.0
      b(2) = 0.0
      b(3) = 0.49
      b(4) = 0.49
c                                                         { Shrink viewport 1. }
      call gm_$viewport_set_bounds
     +      ( b
     +      , st
     +      )

      b(1) = 0.51
      b(2) = 0.0
      b(3) = 1.0
      b(4) = 0.49
c                                                         { Create viewport 2. }
      call gm_$viewport_create
     +      ( b
     +      , vpid2
     +      , st
     +      )

      b(1) = 0.0
      b(2) = 0.51
      b(3) = 0.49
      b(4) = 1.0
c                                              { Create viewport 3. }
      call gm_$viewport_create
     +      ( b
     +      , vpid3
     +      , st
     +      )

      b(1) = 0.51
      b(2) = 0.51
      b(3) = 1.0
      b(4) = 1.0
c                                              { Create viewport 4. }
      call gm_$viewport_create
     +      ( b
     +      , vpid4
     +      , st
     +      )

c                                           { Red + green = yellow. }
c      FOR k = 8 TO 15
c      DO WITH color_array[ k ]
c      DO BEGIN
c            red   = 1.0
c            green = 1.0
c            blue  = 0.0
c            END}
c
c                             { Set color values 8 to 15 to yellow. }
      call gm_$display_set_color_map
     +      ( int4(8)
     +      , int2(8)
     +      , color_array
     +      , st
     +      )

c                                               { Create an ablock. }
      call gm_$ablock_create
     +      ( int2(1)
     +      , ablockid
     +      , st
     +      )

c                          { For the ablock, set the draw value to 9. }
      call gm_$ablock_set_draw_value
     +      ( ablockid
     +      , int4(9)
     +      , st
     +      )

c                          { Assign the ablock to the default aclass. }
      call gm_$ablock_assign_display
     +       ( gm_default_aclass
     +       , ablockid
     +       , st
     +       )

c                                         { Display 'grid' in viewport 4. }
      call gm_$display_file
     +       ( st
     +       )

c                                                     { Wait a moment. }
       call wait

c                                  { Reset the ablock to default attributes. }
      call gm_$ablock_copy
     +       ( int2(1)
     +       , ablockid
     +       , st
     +       )

c                           { For the ablock pass the number 7. This turns the }
c                       { first three bits on in the word. Each bit corresponds }
c                                                          { to a plane. }
      call gm_$ablock_set_plane_mask
     +       ( ablockid
     +       , .TRUE.
     +       , int2(7)
     +       , st
     +       )

c                                               { Create segment 'box.' }
      call gm_$segment_create
     +       ( 'box'
     +       , int2(3)
     +       , sid2
     +       , st
     +       )

      pt1(1) = 0
      pt1(2) = 0
      pt2(1) = 10
      pt2(2) = 10
c                                            { Add a rectangle to 'box.' }
      call gm_$rectangle_16
     +       ( pt1
     +       , pt2
     +       , .FALSE.
     +       , st
     +       )

c                                                { Close segment 'box.' }
      call gm_$segment_close
     +       ( .TRUE.
     +       , st
     +       )

c                                               { Create segment 'row.' }
      call gm_$segment_create
     +       ( 'row'
     +      , int2(3)
     +      , sid3
     +      , st
     +      )

c                              { Instance segment 'box' into segment 'row.' }
      transl(1) = 0
      transl(2) = 0
      DO 200 k = 1,repeats
            transl(1) = transl(1) + space
            call gm_$instance_translate_2d16
     +            ( sid2
     +            , transl
     +            , st
     +            )
200   continue

c                                                  { Close segment 'row.' }
      call gm_$segment_close
     +      ( .TRUE.
     +      , st
     +      )

c                                               { Create segment 'block.' }
      call gm_$segment_create
     +      ( 'block'
     +      , int2(5)
     +      , sid4
     +      , st
     +      )

c                            { Instance segment 'row' into segment 'block.' }
      transl(2) = 50
      DO 300 k = 1,repeats
            transl(1) = k
            transl(2) = transl(2) - space
            call gm_$instance_translate_2d16
     +            ( sid3
     +            , transl
     +            , st
     +            )
300   continue

c                                                { Close segment 'block.' }
      call gm_$segment_close
     +      ( .TRUE.
     +      , st
     +      )

c                                 { Display segment 'block' in viewport 3. }
      call gm_$display_segment
     +      ( sid4
     +      , st
     +      )

c                                                       { Wait a moment. }
       call wait

      rtransl(1) = 0.5
      rtransl(2) = 1.0
c                                         { For viewport 3, zoom out. }
      call gm_$view_scale
     +      ( 0.25
     +      , rtransl
     +      , st
     +      )

      rtransl(1) = -0.06
      rtransl(2) = 0.0
      DO 400 k = 1,5
c                             { For viewport 3, pan from left to right. }
      call gm_$view_translate
     +      ( rtransl
     +      , st
     +      )
400   continue

      rtransl(1) = 0.5
      rtransl(2) = 0.5
      DO 500 k = 1,5
c                    { For viewport 3, pan diagonally towards lower left. }
      call gm_$view_scale
     +      ( 0.85
     +      , rtransl
     +      , st
     +      )
500   continue

c                                                    { Wait a moment. }
      call wait
c                       { For viewport 2, set the background value to 2. }

      call gm_$viewport_set_background_valu
     +      ( vpid2
     +      , int4(2)
     +      , st
     +      )

c                                                { Select viewport 2. }
      call gm_$viewport_select
     +      ( vpid2
     +      , st
     +      )

c                               { Display segment 'row' in viewport 2. }
      call gm_$display_segment
     +      ( sid3
     +      , st
     +      )

c                                                    { Wait a moment. }
       call wait

c                          { For viewport 3, set background value to 2. }
      call gm_$viewport_set_background_valu
     +      ( vpid3
     +      , int4(3)
     +      , st
     +      )

c                                               { Select viewport 3. }
      call gm_$viewport_select
     +      ( vpid3
     +      , st
     +      )

c                             { Display segment 'block' in viewport 3. }
      call gm_$display_segment
     +      ( sid4
     +      , st
     +      )

c                                                  { Wait a moment. }
       call wait

c                                                  { Close the file. }
      call gm_$file_close
     +      ( .TRUE.
     +      , st
     +      )

c                                       { Terminate the 2D GMR package. }
      call gm_$terminate
     +      ( st
     +      )

      END

      subroutine check
     +      ( status
     +      )
      integer*4          status

      IF (status .ne. 0)THEN
      call pfm_$error_trap( status )
      ENDIF

      return
      end

      subroutine wait

      integer*2               pause(3)
      integer*4               status

      pause(1) = 0
      pause(2) = 20
      pause(3) = 0
c     { Wait five seconds. }
            call time_$wait
     +            ( time_$relative
     +            , pause
     +            , status
     +            )
      call check( status )
      END
```

Attributes and Instancing

The following program displays a file as it is being created and edited. The file creates the picture in Figure D-1.

```
      PROGRAM hotel

%INCLUDE '/sys/ins/base.ins.ftn'
%INCLUDE '/sys/ins/error.ins.ftn'
%INCLUDE '/sys/ins/pfm.ins.ftn'
%INCLUDE '/sys/ins/gmr.ins.ftn'
%INCLUDE '/sys/ins/time.ins.ftn'


      integer*4          status
      integer*2          file_id
      integer*2          font_file_id
      integer*4          sid_scene
      integer*4          sid_door
      integer*4          sid_window
      integer*4          sid_sign
      integer*4          sid_tree
      integer*4          sid_house
      character          pattern(8)
      integer*2          p(12)
      integer*2          q(10)
      integer*2          center(2)
      integer*2          radius
      integer*2          i
      integer*2          pause(3)
      integer*2          corner1(2),corner2(2)
      integer*2          bitmap_size(2)


c                                        { Intialize 2D GMR package. }
      bitmap_size(1) = 1024
      bitmap_size(2) = 1024
      call GM_$INIT
     +      ( gm_$direct
     +      , stream_$stdout
     +      , bitmap_size
     +      , int2(8)
     +      , status
     +      )
      call check(status)

c                                        { Create and name a metafile. }
      call GM_$FILE_CREATE
     +      ( 'hotel.gm'
     +      , int2(8)
     +      , gm_$overwrite
     +      , gm_$1w
     +      , file_id
     +      , status
     +      )
      call check(status)
```

```
c                                                 { Load a font family. }
      call GM_$FONT_FAMILY_include
     +      ( 'font_families'
     +      , int2(13)
     +      , gm_$pixel
     +      , font_file_id
     +      , status
     +      )
      call check(status)

c                                                { Set the data coerce. }
      call GM_$DATA_COERCE_SET_REAL
     +      ( gm_$32
     +      , status
     +      )
      call check(status)

c                                    { Create the segment for the door. }
      call GM_$SEGMENT_CREATE
     +      ( ''
     +      , int2(0)
     +      , sid_door
     +      , status
     +      )
      call check(status)

c                                                  { Construct the door. }
      corner1(1) = 0
      corner1(2) = 0
      corner2(1) = 36
      corner2(2) = 80

      call GM_$RECTANGLE_16
     +      ( corner1
     +      , corner2
     +      , .TRUE.
     +      , status
     +      )
      call check(status)

c                                             { Construct the door knob. }
      call GM_$FILL_VALUE
     +      ( int4(0)
     +      , status
     +      )
      call check(status)

      corner1(1) = 30
      corner1(2) = 38
      corner2(1) = 33
      corner2(2) = 41
      call GM_$RECTANGLE_16
     +      ( corner1
     +      , corner2
     +      , .TRUE.
     +      , status
     +      )
      call check(status)
      call GM_$SEGMENT_CLOSE
     +      ( .TRUE.
     +      , status
     +      )
      call check(status)

c                                    { Create the segment for the window. }
      call GM_$SEGMENT_CREATE
     +      ( ''
     +      , int2(0)
     +      , sid_window
     +      , status
     +      )
      call check(status)

      corner1(1) = 0
      corner1(2) = 0
      corner2(1) = 36
      corner2(2) = 36
      call GM_$RECTANGLE_16
     +      ( corner1
     +      , corner2
     +      , .FALSE.
     +      , status
     +      )
      call check(status)

      p(1) = 0
      p(2) = 18
      p(3) = 36
      p(4) = 18
      call GM_$POLYLINE_2D16
     +      ( int2(2)
     +      , p
     +      , .FALSE.
     +      , .FALSE.
     +      , status
     +      )
      call check(status)

      p( 1 ) = 18
      p( 2 ) = 0
      p( 3 ) = 18
      p( 4 ) = 36
      call GM_$POLYLINE_2D16
     +      ( int2(2)
     +      , p
     +      ,.FALSE.
     +      ,.FALSE.
     +      , status
     +      )
      call check(status)

      call GM_$SEGMENT_CLOSE
     +      ( .TRUE.
     +      , status
     +      )
      call check(status)
```

```
c                                   { Create the segment for the sign. }
      call GM_$SEGMENT_CREATE
     +      ( ''
     +      , int2(0)
     +      , sid_sign
     +      , status
     +      )
      call check(status)

      call GM_$TEXT_SIZE
     +      ( 14.0
     +      , status
     +      )
      call check(status)

      p( 1 ) = 0
      p( 2 ) = 0
      call GM_$TEXT_2D16
     +      ( p
     +      , 0.0
     +      , 'GRAND MOTEL'
     +      , int2(11)
     +      , status
     +      )
      call check(status)

      call GM_$SEGMENT_CLOSE
     +      ( .TRUE.
     +      , status
     +      )
      call check(status)

c                                   { Create the segment for the house. }
      call GM_$SEGMENT_CREATE
     +      ( ''
     +      , int2(0)
     +      , sid_house
     +      , status
     +      )
      call check(status)

c                                                   { Build the house. }
      corner1( 1 ) = 0
      corner1( 2 ) = 0
      corner2( 1 ) = 480
      corner2( 2 ) = 260
      call GM_$RECTANGLE_16
     +      ( corner1
     +      , corner2
     +      , .FALSE.
     +      , status
     +      )
      call check(status)

c                                                    { Build the roof. }
      p(1) = -10
      p(2) = 255
      p(3) = 240
      p(4) = 380
```

```
       p(5) = 490
       p(6) = 255

       call GM_$POLYLINE_2D16
     +       ( int2(3)
     +       , p
     +       , .FALSE.
     +       , .FALSE.
     +       , status
     +       )
       call check(status)

c                                                 { Build the chimney. }
       p(1) = 300
       p(2) = 350
       p(3) = 300
       p(4) = 370
       p(5) = 330
       p(6) = 370
       p(7) = 330
       p(8) = 335
       call GM_$POLYLINE_2D16
     +       ( int2(4)
     +       , p
     +       , .FALSE.
     +       , .FALSE.
     +       , status
     +       )
       call check(status)

c                                            { Build the round window. }
       center(1) = 240
       center(2) = 195
       radius = 45

       call GM_$CIRCLE_16
     +       ( center
     +       , radius
     +       , .FALSE.
     +       , status
     +       )
       call check(status)

       p(1) = center(1) - radius
       p(2) = center(2)
       p(3) = center(1)
       p(4) = center(2) + radius
       p(5) = center(1) + radius
       p(6) = center(2)
       p(7) = center(1)
       p(8) = center(2) - radius
c                                      { Draw rectangle inside window. }

       call GM_$POLYLINE_2D16
     +       ( int2(4)
     +       , p
     +       , .TRUE.
     +       , .FALSE.
     +       , status
     +      )
      call check(status)

      p( 1 ) = center(1) - radius
      p( 2 ) = center(2)
      p( 3 ) = center(1) + radius
      p( 4 ) = center(2)

c                                  { Draw horizontal line inside window. }

      call GM_$POLYLINE_2D16
     +      ( int2(2)
     +      , p
     +      , .FALSE.
     +      , .FALSE.
     +      , status
     +      )
      call check(status)

      p( 1 ) = center(1)
      p( 2 ) = center(2) + radius
      p( 3 ) = center(1)
      p( 4 ) = center(2) - radius


      call GM_$POLYLINE_2D16
     +      ( int2(2)
     +      , p
     +      , .FALSE.
     +      , .FALSE.
     +      , status
     +      )
      call check(status)

c                                      { Instance and position the door. }
      q( 1 ) = 222
      q( 2 ) = 0
      call GM_$INSTANCE_TRANSLATE_2D16
     +      ( sid_door
     +      , q
     +      , status
     +      )
      call check(status)

c                                   { Instance and position the windows. }
      q( 1 ) = 50
      q( 2 ) = 40
      call GM_$INSTANCE_TRANSLATE_2D16
     +      ( sid_window
     +      , q
     +      , status
     +      )
      call check(status)

      q( 1 ) = 118
      call GM_$INSTANCE_TRANSLATE_2D16
     +      ( sid_window
     +      , q
     +      , status
     +       )
      call check(status)

      q( 1 ) = 326
      call GM_$INSTANCE_TRANSLATE_2D16
     +       ( sid_window
     +       , q
     +       , status
     +       )
      call check(status)

      q( 1 ) = 394
      call GM_$INSTANCE_TRANSLATE_2D16
     +       ( sid_window
     +       , q
     +       , status
     +       )

      q( 2 ) = 180
      call GM_$INSTANCE_TRANSLATE_2D16
     +       ( sid_window
     +       , q
     +       , status
     +       )

      q( 1 ) = 326
      call GM_$INSTANCE_TRANSLATE_2D16
     +       ( sid_window
     +       , q
     +       , status
     +       )
      call check(status)

      q( 1 ) = 118
      call GM_$INSTANCE_TRANSLATE_2D16
     +       ( sid_window
     +       , q
     +       , status
     +       )
      call check(status)

      q( 1 ) = 50
      call GM_$INSTANCE_TRANSLATE_2D16
     +       ( sid_window
     +       , q
     +       , status
     +       )
      call check(status)

c                            { Instance and position the segment for the sign. }
      q( 1 ) = 172
      q( 2 ) = 120
      call GM_$INSTANCE_TRANSLATE_2D16
     +       ( sid_sign
     +       , q
     +       , status
     +       )
      call check(status)
      call GM_$SEGMENT_CLOSE
     +       ( .TRUE.
     +       , status
     +       )
      call check(status)

c                                        { Create the segment for the trees. }
      call GM_$SEGMENT_CREATE
     +       ( ''
     +       , 0
     +       , sid_tree
     +       , status
     +       )
      call check(status)

      p( 1 ) = 0
      p( 2 ) = 0
      p( 3 ) = 0
      p( 4 ) = 150
      call GM_$POLYLINE_2D16
     +       ( int2(2)
     +       , p
     +       , .FALSE.
     +       , .FALSE.
     +       , status
     +       )
      call check(status)

      p( 1 ) = 12
      p( 3 ) = 12
      call GM_$POLYLINE_2D16
     +       ( int2(2)
     +       , p
     +       , .FALSE.
     +       , .FALSE.
     +       , status
     +       )
      call check(status)

      p( 1 ) = 6
      p( 2 ) = 200
      call GM_$CIRCLE_16
     +       ( p
     +       , int2(50)
     +       , .FALSE.
     +       , status
     +       )
      call check(status)

      call GM_$DRAW_STYLE
     +       ( gm_$dotted
     +       , int2(2)
     +       , pattern
     +       , int2(0)
     +       , status
     +       )

      p( 1 ) = 0
      p( 2 ) = 180
```

```
 p( 3 ) = -40
 p( 4 ) = 200
 call GM_$POLYLINE_2D16
+      ( int2(2)
+      , p
+      , .FALSE.
+      , .FALSE.
+      , status
+      )

 p( 1 ) = 12
 p( 3 ) = 52
 call GM_$POLYLINE_2D16
+      ( int2(2)
+      , p
+      , .FALSE.
+      , .FALSE.
+      , status
+      )
 call check(status)

 p( 1 ) = 4
 p( 2 ) = 190
 p( 3 ) = -20
 p( 4 ) = 230
 call GM_$POLYLINE_2D16
+      ( int2(2)
+      , p
+      , .FALSE.
+      , .FALSE.
+      , status
+      )
 call check(status)

 p( 1 ) = 8
 p( 2 ) = 190
 p( 3 ) = 32
 p( 4 ) = 230
 call GM_$POLYLINE_2D16
+      ( int2(2)
+      , p
+      , .FALSE.
+      , .FALSE.
+      , status
+      )
 call check(status)

 p( 1 ) = 6
 p( 2 ) = 195
 p( 3 ) = 6
 p( 4 ) = 240
 call GM_$POLYLINE_2D16
+      ( int2(2)
+      , p
+      , .FALSE.
+      , .FALSE.
+      , status
+      )
 call check(status)
```

```
      p( 1 ) = 0
      p( 2 ) = 170
      p( 3 ) = 0
      p( 4 ) = 150
      call GM_$POLYLINE_2D16
     +       ( int2(2)
     +       , p
     +       , .FALSE.
     +       , .FALSE.
     +       , status
     +       )
      call check(status)

      p( 1 ) = 12
      p( 3 ) = 12
      call GM_$POLYLINE_2D16
     +       ( int2(2)
     +       , p
     +       , .FALSE.
     +       , .FALSE.
     +       , status
     +       )
      call check(status)

      call GM_$SEGMENT_CLOSE
     +       ( .TRUE.
     +       , status
     +       )
      call check(status)

c                                  { Create the segment called scene. }
      call GM_$SEGMENT_CREATE
     +       ( ''
     +       , int2(0)
     +       , sid_scene
     +       , status
     +       )
      call check(status)

c                               { Instance the segment for the house. }
      p( 1 ) = 0
      p( 2 ) = 0
      call GM_$INSTANCE_TRANSLATE_2D16
     +       ( sid_house
     +       , p
     +       , status
     +       )
      call check(status)

c             { Instance, translate, and scale the segment for the trees. }
      p( 1 ) = -85
      p( 2 ) = -25
      call GM_$INSTANCE_SCALE_2D16
     +       ( sid_tree
     +       , 2.0
     +       , p
     +       , status
     +       )
```

```
      p( 1 ) = 530
      p( 2 ) = 55
      call GM_$INSTANCE_SCALE_2D16
     +      ( sid_tree
     +      , 0.75
     +      , p
     +      , status
     +      )
      call check(status)

      p( 1 ) = 610
      p( 2 ) = 105
      call GM_$INSTANCE_SCALE_2D16
     +      ( sid_tree
     +      , 0.85
     +      , p
     +      , status
     +      )
      call check(status)

      call GM_$SEGMENT_CLOSE
     +      ( .TRUE.
     +      , status
     +      )
      call check(status)

c                                  { Now display the completed scene. }
      call GM_$DISPLAY_SEGMENT
     +      ( sid_scene
     +      , status
     +      )
      call check(status)

c                                    { Admire the scene for five seconds. }
      pause(1) = 0
      pause(2) = 20
      pause(3) = 0
      call TIME_$WAIT
     +      ( time_$relative
     +      , pause
     +      , status
     +      )
      call check(status)

c                                            { Close and save the file. }
      call GM_$FILE_CLOSE
     +      ( .TRUE.
     +      , status
     +      )
      call check(status)

c                                           { Terminate the GMR package. }
      call GM_$TERMINATE
     +      ( status
     +      )
      call check(status)

      END
```

```
c*********************************************************************
      subroutine check(status)
      integer*4 status

%include '/sys/ins/base.ins.ftn'
%include '/sys/ins/pfm.ins.ftn'



            IF (status .ne. 0)then
            call pfm_$error_trap(status)
            endif
      return
      end
```

Index

READER'S RESPONSE

We use readers' comments in revising and improving our documents.

Document Title: *Programming With DOMAIN 2D Graphics Metafile Resource*
Order Number: 005097
Revision: 00
Date of Publication: July, 1985

What is the best feature of this manual?

____
____
____

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible.)

____
____
____
____

What type of user are you?

______ Systems programmer; language ______________________________
______ Applications programmer; language __________________________
______ Manager/Professional
______ Technical Professional
______ Adminstrative/Support Personnel
______ Student programmer
______ User with little programming experience
______ Other

How often do you use your system?

____

Nature of your work on the DOMAIN System:

____

____
Your name   Date

____
Organization

____
Street Address

____
City   State   Zip/Country

No postage necessary if mailed in the U.S. Fold on dotted lines (see reverse), tape, and mail.

cut or fold along dotted line

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

APOLLO COMPUTER INC.
Technical Publications
P.O. Box 451
Chelmsford, MA 01824

FOLD