```
process AIRPLANE
     call TOWER giving GATE yielding RUNWAY
     work TAXI.TIME (GATE, RUNWAY) minutes
     request 1 RUNWAY
     work TAKEOFF.TIME (AIRPLANE) minutes
     relinquish 1 RUNWAY
end " process AIRPL

 process AIRP
      call T
      work
      requ
```

SIMGRAPHICS II
User's Manual

CACI
Products Company

For product information or technical support contact:

In the US and Pacific Rim:

CACI Products Company
3333 North Torrey Pines Court
La Jolla, California 92037
Phone: (619) 824.5200
Fax: (619) 457.1184

In Europe:

CACI Products Division
Coliseum Business Centre
Riverside Way, Camberley
Surrey GU15 3YL UK
Phone: +44 (0) 1276.671.671
Fax: +44 (0) 1276.670.677

Contents

Preface ..... **a**

Why Use Graphics? ..... b
Organization of This Manual ..... b

1. Overview of SIMGRAPHICS II **1**

1.1 Effective Use of Graphics and the User Interface 1

1.1.1 Selecting Colors 2
1.1.2 Scale and Size 2
1.1.3 Designing a Background 2
1.1.4 Representing Changes in System State 3
1.1.5 How Many Objects Should Be Displayed? 3

2. Tutorial **5**

2.1 How to Open a SIMGRAPHICS II Window with a Title 5
2.2 Display Icons in the Default Window 6
2.3 Use of Multiple Graphics Libraries 7
2.4 Example "Window" 7
2.5 How to Open Multiple SIMGRAPHICS II Windows 9
2.6 SIMDRAW — the Graphics Editor 11
2.7 Creating an Icon 14
2.8 Adding Animation 15
2.9 Creating a Dialog Box 17
2.10 Adding Graphical User Interaction Using Dialog Boxes 19
2.11 Creating a Graph 21
2.12 Adding Presentation Graphics 21
2.13 Creating a PostScript File 23
2.14 Using a Bitmap as a Background 23
2.15 Creating Cascadeable Menus 26
2.16 Using Cascadeable Menus 27

2.16.1 Cascadeable Menus in Simulation Programs 27

3. SIMDRAW **31**

3.1 SIMDRAW Overview 31
3.2 Running SIMDRAW 31
3.3 Loading and Saving SIMGRAPHICS II Files 32
3.4 Editing an Existing Object 32
3.5 Adding an Object to the Library 33
3.6 Removing an Object from the Library 33
3.7 Making a Duplicate of an Object 33
3.8 Changing the Name of an Object 33
3.9 Adding an Object from Another Library 33
3.10 Editing Images and Graphs in Same Window 33
3.11 User Preferences 33
3.12 Command Line Arguments 34
3.13 Using the Image Editor 35

3.13.1 Mode, Style, and Color Palettes 35
3.13.2 Selecting, Moving, and Resizing 36
3.13.3 Using the Clipboard (Cut, Copy, Paste Commands) 36
3.13.4 Importing / Exporting to Other Graphical Formats 36
3.13.5 Creating Primitives 37
3.13.6 Creating Images 40

3.13.7 Editing the Root Image 41
3.13.8 Editing Points on a Primitive 41
3.13.9 Defining Stacking Order or Priority 41
3.13.10 Defining the Center Point of a Shape 42
3.13.11 Using the Flip and Rotate Tools 42
3.13.12 Align and Distribute 42
3.13.13 Using Grid Lines 43
3.13.14 Changing Views (Panning and Zooming) 43
3.13.15 Changing Dimension (Coordinate Space Boundaries) 43
3.13.16 Changing the Layout Size and Color 44
3.13.17 Program Access 44

3.14 USING THE GRAPH EDITOR 44

3.14.1 Style, and Color Palettes 44
3.14.2 Selecting, Moving, and Resizing 45
3.14.3 Charts (2-D Plots) 45
3.14.4 Pie Charts 49
3.14.5 Clocks 50
3.14.6 Dials 51
3.14.7 Level Meters 51
3.14.8 Digital Displays 52
3.14.9 Text Meters 52

3.15 USING THE DIALOG EDITOR 52

3.15.1 Selecting, Moving, and Resizing 53
3.15.2 Dialog Box Coordinate System 53
3.15.3 Using the Clipboard (Cut, Copy, Paste Commands) 54
3.15.4 Controls 54

3.16 USING THE MENU BAR EDITOR 62

3.16.1 Selecting and Moving (Transferring) 64
3.16.2 Using the Clipboard (Cut, Copy and Paste Commands) 64
3.16.3 Editing the Menu Bar 64
3.16.4 Editing a Menu 65
3.16.5 Editing a Menu Item 65

3.17 USING THE PALETTE EDITOR 66

3.17.1 Selecting and Moving (Rearrangement of) Buttons 67
3.17.2 Using the Clipboard (Cut, Copy and Paste) 68
3.17.3 Editing the Palette 68
3.17.4 Editing a Palette Button 69
3.17.5 Editing Palette Separators 69

4. Creating Presentation Graphics 71

4.1 VARIABLE DECLARATION 71
4.2 DISPLAYING PRESENTATION GRAPHICS 72
4.3 EXAMPLES 73

4.3.1 Example 1: A Simple Tallied Histogram 73
4.3.2 Example 2: A Time-Weighted Accumulated Dynamic Histogram 74
4.3.3 Example 3: Displaying Simple Scalar Values 75
4.3.4 Example 4: Using a Trace to Plot X-Y Curves 76
4.3.5 Example 5: The Bank Model 77

5. Forms and Graphical Interaction 79

5.1 INTRODUCTION 79

5.2 Creating a Form ... 80
5.2.1 Reference Names and Field Attributes ... 81
5.3 Using the Form in a Program ... 82
5.3.1 Using ACCEPT.F ... 82
5.3.2 Interaction Modes ... 83
5.4 Field Attributes ... 84
5.4.1 Value Attributes ... 84
5.4.2 Terminating Buttons ... 85
5.4.3 Verifying Buttons ... 85
5.5 Form Control Routines ... 85
5.6 Details of Field Operations ... 86
5.6.1 The DISPLAY Command ... 86
5.6.2 The **ACCEPT.F** Function ... 86
5.6.3 The ERASE Command ... 86
5.6.4 The DESTROY Command ... 87
5.6.5 The SET.ACTIVATION.R Routine ... 87
5.7 Dialog Boxes and Their Fields ... 87
5.7.1 Dialog Box ... 87
5.8 Predefined Dialog Boxes ... 92
5.8.1 Standard Message Dialog ... 92
5.8.2 Custom Message Dialogs (Alert, Stop, Information and Question) ... 92
5.8.3 File Selection Dialog ... 93
5.8.4 System Font Browser ... 93
5.8.5 Printing the Contents of a Graphics Window (or Individual Segment) ... 94
5.9 Menu Bars and Palettes ... 94
5.9.1 Menu Bar ... 94
5.9.2 Palettes ... 95
5.10 Examples ... 96
6. Creating Animated Graphics ... 101
6.1 Graphic Entity Declaration ... 102
6.2 Coordinate Systems ... 102
6.2.1 Normalized Device Coordinates ... 103
6.2.2 Setting a Viewing Transformation ... 103
6.2.3 Defining The World: **SETWORLD.R** ... 104
6.2.4 Defining a Viewport: **Routine SETVIEW.R** ... 105
6.2.5 Modelling Transformations ... 105
6.3 Animating Dynamic Graphic Entities ... 107
6.4 Displaying Icons ... 107
6.5 An Example ... 108
6.5.1 Preamble ... 108
6.5.2 Main Program ... 108
6.5.3 Process Shape ... 109
6.6 Destroying and Erasing Icons ... 110
6.7 Synchronizing Simulation Time and Real Time ... 110
7. Example Programs ... 113
7.1 The Gold Mine Program ... 113

7.1.1 Menu Bar Process 114
7.1.2 Form Control Routine 114
7.2 The DYNHIST Model 117
7.3 The Port Model 118
7.4 The CALSHIP Model 119
7.5 The Spring Model 120
7.6 The Pilot Ejection Model 121
8. Managing Multiple Windows 123
8.1 Multiple Window Support 123
8.2 Setting and Getting the Attributes and Events of a Window 124
8.2.1 Window Attributes or "Fields" 124
8.3 Window Events 126
8.4 Scrollable Windows 127
8.5 Status Bars 128
9. Advanced Topics 131
9.1 Drawing Icons Without SIMDRAW 131
9.2 Writing a Display Routine 131
9.2.1 Color 132
9.2.2 Drawing Areas 132
9.2.3 Drawing Lines 133
9.2.4 Drawing Points (Markers) 134
9.2.5 Direct Character Output 134
9.2.6 Character Output Using System Text 135
9.2.7 System Font Browser 135
9.2.8 Loading a Font Re-definition File 136
9.2.9 The Shape Example Revisited 136
9.3 Using Segments 138
9.3.1 Segment Priority 139
9.3.2 Using Priority Zero 139
9.3.3 Other Segment Operations 139
9.3.4 Drawing Backgrounds 140
9.4 Additional Attributes of [Dynamic] Graphic Entities 140
9.5 Low-Level Input Constructs 141
9.5.1 Selecting a Segment 142
9.6 Programmatically Definable System Cursor 142
9.7 Time Unit Conversion for Simulation Graphics 142
Appendix A. SIMGRAPHICS II Variables and Routines 145
Appendix B. Conversion to SIMGRAPHICS II 171
B.1 What is SIMGRAPHICS II? 171
B.2 Differences Between SIMGRAPHICS I and II 171
B.2.1 Icons 172
B.2.2 Graphs 172
B.2.3 Forms 172
B.2.4 Menu Bars 172
B.2.5 Dialog Boxes 173
B.2.6 Push Buttons 173
B.2.7 Radio Buttons 173

B.3 USING THE CONVERSION UTILITY 173
B.3.1 Calling SIMCVT — Command Line Arguments 173
B.3.2 Possible Problems with Forms 174
B.3.3 A Menu Bar Within a Form 174
B.3.4 Conversion of Files from PC DOS SIMSCRIPT 175
B.3.5 Miscellaneous Notes 175
B.3.6 Features No Longer Supported in SIMGRAPHICS II 175
Index 177

List of Figures

Figure 2-1. Example Window 8
Figure 2-2. Multiple Window Example 10
Figure 2-3. SIMDRAW Main Window 11
Figure 2-4. Creating a Cart Icon 15
Figure 2-5. Output of the Image-1 Routine 16
Figure 2-6. Dialog Box Editor Window 18
Figure 2-7. Dialog Box for Example IMAGE 2 19
Figure 2-8. Example IMAGE-3 21
Figure 2-9. Example “San Diego” Showing Imported Bitmap 24
Figure 2-10. Example of a Bitmap Used as a Background 26
Figure 2-11. Cascadeable Menu 28
Figure 3-1. Main Window 32
Figure 3-2. Image Editor 35
Figure 3-3. Dialog Editor 53
Figure 3-4. Menu Bar Editor 63
Figure 3-5. Palette Editor 67
Figure 4-1. Example 1 73
Figure 4-2. Example 2 74
Figure 4-3. Example 3 75
Figure 4-4. Example 4 76
Figure 4-5. The Bank Model 77
Figure 5-1. Form for the ATM Example 97
Figure 5-2. Form for List1 Example 97
Figure 6-1. Animated Icons 102
Figure 6-2. Coordinate Transformations 103
Figure 6-3. Object Origin 106
Figure 6-4. Output of the Shape Routine 108
Figure 7-1. The Gold Mine 114
Figure 7-2. Output of the DYNHIST Model 117
Figure 7-3. The Port Model 118
Figure 7-4. The CALSHIP Model 119
Figure 7-5. The Spring Model 120
Figure 7-6. The EJECT Model 121

Preface

Over the past few years interactive colored graphics based on windows systems have become standard on every personal computer and workstation. These graphical systems are provided by computer system vendors and all have different programming interfaces. There is no standard, so graphical programs developed on a PC platform using the Windows toolkit cannot easily be ported to a UNIX workstation without changes in source code, and vice versa. In addition, vendor's toolkits change constantly and are not easy to use. Graphical programs which use them directly are not portable and are difficult to maintain.

Following our tradition of developing easy-to-use, stable and reliable products, pioneering new concepts and exploiting proven state-of-the-art technologies, CACI has developed SIMGRAPHICS II©, the second generation of CACI's SIMGRAPHICS.

SIMGRAPHICS II© preserves all the best, well tested, and proven concepts from its predecessor: the same programming interface, high expressive power**,** simplicity, ease of use and portability across various platforms.

Programs written in SIMSCRIPT II.5 with SIMGRAPHICS II are portable. They can run on another platform without any source code changes. And, because SIMGRAPHICS II uses underlying system vendor toolkits, graphical SIMSCRIPT II.5 programs automatically acquire the look-and-feel of the platform they are running on without any program modifications.

To preserve your investments in program development, SIMGRAPHICS II was designed with forward and backward compatibility in mind. It is superset of SIMGRAPHICS I and graphical programs written in SIMGRAPHICS I will run using SIMGRAPHICS II with only minor changes.

SIMGRAPHICS II is a self-contained, state-of-the-art graphical package interfaced with SIMSCRIPT II.5. It provides all the necessary tools, language statements and data structures for creating new interactive graphical programs, or adding an interactive graphical interface, static or dynamic presentation graphics and animation graphics to existing SIMSCRIPT II.5 models.

It also provides automatic generation of encapsulated PostScript output from the running models for hard copy documentation.

SIMGRAPHICS II provides support for bitmaps, allowing geographic maps, street maps, airport layouts, etc., to be used as a background, to create realistic presentations and improve the accuracy of simulated environments.

Synchronous presentation in multiple windows allows the modeler to better structure visual information and graphical interactions.

Why Use Graphics?

The goal of a system simulation is to increase the understanding of the operation of a complex system. Unfortunately, the results of simulation studies are often presented only as pages of numbers, which fail to communicate the understanding gained. The complexity of the system and the simulation can make it difficult for users and decision makers to fully appreciate the interactions between system elements. Results are often not well understood. Premature action may be taken based on invalid assumptions, incorrect data, or hidden modeling errors. Conversely, valid simulation results may not be quickly appreciated.

Often the best way to represent a dynamic system is graphically. Animated graphics clearly show the operation of the simulated system and graphic results are easily evaluated. System operation is better understood, and decision makers have more confidence in the simulation results.

Graphical representation facilitates debugging. Coding, data and modeling errors are apparent, thus avoiding the need for tedious error tracking.

SIMSCRIPT II.5 has a wide range of applications, and SIMGRAPHICS II is flexible enough to represent entities ranging from aircraft on runways to messages in a communications network. It is easy to use. All graphics features are part of a self-documenting SIMSCRIPT II.5 model program. Control of the simulation is separated from the running of the animation. Animated graphical output can easily be added to existing models.

Organization of This Manual

This manual is organized to let you add interactive colored graphics to your SIMSCRIPT II.5 applications almost immediately. The fundamental concepts of SIMGRAPHICS II are covered in Chapter 1. A brief tutorial on SIMGRAPHICS II is presented in Chapter 2. SIMDRAW is described in detail in Chapter 3. Presentation graphics, graphical interactions through forms, and animated graphics are described in Chapters 4, 5, and 6. These chapters use a cookbook approach and include all the information necessary to create most kinds of animated displays. All three types of graphics can be included in a single simulation simultaneously, and examples of simulations including each kind of graphics are given in Chapter 7. Managing multiple windows is described in Chapter 8. Chapter 9 covers advanced topics such as the programmatic approach for the manipulation of graphic images, real-time synchronization, etc. Appendix A is a reference to all of SIMGRAPHICS II's variables and routines. Appendix B is a guide to converting SIMGRAPHICS I programs to SIMGRAPHICS II.

Additional information regarding the behavior of a specific computer system is contained in the *SIMSCRIPT II.5 User's Manual* for that system.

For information on the free trial of SIMSCRIPT II.5 with SIMGRAPHICS II, or SIMSCRIPT II.5 documentation or books, contact:

In the U.S. & Pacific Rim:

CACI Products Company
3333 North Torrey Pines Court
La Jolla, CA 92037
(619) 824-5200
Fax (619) 457-1184

In Europe:

CACI Products Division
Coliseum Business Centre
Riverside Way, Camberley
Surrey GU15 3YL
United Kingdom
+44 (0) 1276 671 671
Fax +44 (0) 1276.670.677

1. Overview of SIMGRAPHICS II

SIMGRAPHICS II lets you easily incorporate presentation graphics, interactive graphics and animation in your SIMSCRIPT II.5 programs. SIMGRAPHICS II uses the system vendors' toolkits so your programs acquire the look and feel of whatever system they are running on.

SIMGRAPHICS II's ease of use comes from a design which separates the appearance of icons from the programming of the entities they represent. You do not blindly program the way your graphics look. You see them as you design them. You can change them at any time without recompiling your program.

Presentation graphics include histograms, pie charts, 2-D graphs and other graphics for the visual display of data. Data accumulated in a SIMSCRIPT II.5 program can be presented by adding just a few statements to the program. Most of the work of formatting the graphics is easily done within SIMDRAW.

Interactive programs require input from the user. SIMGRAPHICS II allows you to design input forms using SIMDRAW. Programming for user interaction is usually very tedious. You must watch for mouse clicks, check the range of data entered on forms, check for the opening and closing menus, etc. SIMGRAPHICS II lets you design these types of interactions painlessly using SIMDRAW. You can design forms that include dialog boxes, value boxes, check boxes, list boxes, buttons and menu bars.

Animation requires the design and implementation of icons. Designing icons is very easy in SIMGRAPHICS II. You simply construct the ones you need from a few basic graphic objects: lines, circles, boxes, areas, sectors and polylines. You have the features of an advanced drawing program at your disposal including the ability to cut and paste objects, group objects, snap to a grid, and so on.

These three types of graphics generated by the editor are stored in a single ASCII file called **`graphics.sg2`**. This file is normally stored in the same subdirectory as their program code. It can be moved between different machines and operating systems without modification.

This graphics file can be easily utilized in an application. The program need only contain concepts familiar to the SIMSCRIPT programmer, such as the `TALLY` or `ACCUMULATE` commands, and simple SIMGRAPHICS II constructs such as `SHOW`, `DISPLAY`, `ERASE`, or **`ACCEPT.F`**.

1.1 Effective Use of Graphics and the User Interface

Over the past few years there has been a revolution in computing. It used to be that all programs were text-based and the results were pages of numbers. Since printouts of numbers are not the most effective method of communicating the results of a program, visual output has become more prevalent.

At first programs were used to drive hard copy output devices such as a plotters, but as computer displays have become more sophisticated (and prices have fallen) graphical displays on the computer screen have become commonplace.

Now, not only is the output from a program displayed graphically, but the whole user interface is becoming graphical. Graphical user interfaces can be more intuitive and easier to use than the command line interface. Input can be greatly simplified with interactive graphics. The user can leave some information as its default, alter other data, and be prompted in each step. SIMGRAPHICS II can even use asynchronous input: input which alters the simulation on the fly. Straight text interactions can almost always be improved upon with graphics.

Animated or graphical output is often the most easily absorbed information. The simulation state and results are usually not conveyed as clearly by plain text or value output.

1.1.1 Selecting Colors

Often, the purpose of animated color graphics is to explain or present the results of a simulation study. To this end, select a color scheme that is harmonious, attractive, and meaningful. Remember that color can be over used. This detracts from the entire presentation.

The background color should provide contrast, drawing attention to important information. Effective background colors complement the main color.

Bright colors such as red, orange, and yellow attract attention. These colors can be used for important objects in a display. Subdued colors can be used for less important objects.

Borders around areas help define them. Either a black border or a white border can help set the area off from the background and enhance the entire picture.

Colored objects will look different when surroundings or background fields are different in color. Remember that the appearance of color is a relative feature of the viewer's perception, not an absolute property of color itself.

1.1.2 Scale and Size

Big displays have more visual impact, so whenever possible, use the full area of the screen, even if objects touch or overlap the edge. Objects that overlap the edge will be clipped, and usually look fine.

Scale drawings tend to make objects look insignificant. Using exact scale, while important with CAD systems, is often inappropriate with animation. The better technique is to exaggerate, cartooning the information. Use size, intensity, color, and motion for emphasis. Important objects should be large, and in bright colors.

1.1.3 Designing a Background

A well-designed background makes an animated display more attractive. It also introduces a viewer to the problem, and can provide information about the display. Static items (walls,

stairs, a factory floor layout, an airfield) help associate the simulation results with the real system they represent. A legend can indicate the meaning of different colors or icons. A plain background can change color to indicate a change from day to night shift, and so on. The purpose of animated graphics is often to give simulation results more concrete meaning, and adding a realistic background can be a great help.

1.1.4 Representing Changes in System State

Graphical objects can convey information about changes in system state by changing color, shape, or through motion. For example, a busy machine could be shown in red, an idle one in green, and a broken one in black. Similarly, a busy machine could be represented by one icon and an idle machine by another. An icon representing a machine under repair could include a maintenance person. Labels can also be added to icons, indicating, for example, the type of machine, the type of part being processed, or position in a queue.

1.1.5 How Many Objects Should Be Displayed?

Overall simplicity helps visual presentation. Too many details make it difficult for the viewer to get the message, or may call attention to the wrong features. Less important objects should be simplified or made smaller.

In simulations which contain many objects, it may be desirable to represent only a fraction of them and to label the icon accordingly. A queue of 1000 parts could be represented by 10 icons, each of which stands for 100 parts. Similarly, a numeral could stand for the number of parts being transported, waiting in line, and so on. Also consider your audience when determining the level of detail to depict. Some users are interested in the minute workings of the simulation, while others will only want to see the general pattern of results.

2. Tutorial

SIMGRAPHICS II is part of SIMSCRIPT II.5. It is simple and easy-to-use. Let us see how to create SIMSCRIPT II.5 graphical programs which use SIMGRAPHICS II.

You simply write your SIMSCRIPT II.5 program using additional graphical statements or routine calls, compile it and link with the SIMGRAPHICS II graphical library. Existence of this library or set of libraries is transparent to you. To link with SIMGRAPHICS II on a PC Windows platform, just declare your application as a **SIMGRAPHICS II Application** in your project options. On UNIX you have only to use the script **simgld** for linking.

Graphical statements and calls to graphical library routines are used to declare graphical entities, to open one or more graphical windows, to show a graphical entity with an icon, to animate its motion or to dynamically display variables which change their values over time using smart icons like: graphs, dynamic bar charts, pie charts, clocks etc. or to accept values through dialog boxes.

Icons, graphs and dialog boxes are graphical elements which you create independently of your program. To create them you use the graphical editor SIMDRAW and then store the elements in a library of graphical elements. The default name for this library is **`graphics.sg2`**, but you can create and use different libraries of graphical elements with arbitrary names as long as they all have the extension **`.sg2`**.

To change the appearance of your program you do not have to do any programming, recompiling or relinking. You change the graphical elements used in your program using SIMDRAW. You then store the elements in your library, and execute your program again.

To follow the examples in this tutorial, we assume that you know how to write SIMSCRIPT II.5 programs and how to compile, link and execute them on the specific platform which you are using. SIMSCRIPT II.5 User Manuals for PC Windows, UNIX or VAX/VMS explain in detail each development environment: compiling, linking and running SIMSCRIPT II.5 models.

In this chapter we will show the basics of creating and using SIMGRAPHICS II programs using simple examples.

All graphics elements appear in the graphics window, so first, we will learn how to open a SIMGRAPHICS II window on the screen, write a title in it and display a few icons.

2.1 How to Open a SIMGRAPHICS II Window with a Title

You do not have to open a graphics window. When you create your SIMSCRIPT II.5 program as a SIMGRAPHICS II application, a **default window without any title will be opened automatically** every time the program is executed.

If you want to open a window with a specific title, size and position you can use the graphics library call **`OPENWINDOW.R`** with given arguments: position, title and mapping factor, to get **`WINDOW.ID`** and pass it to the graphical system using the **`SETWINDOW.R`** routine.

By default, the background of the window will be black. If you want to change the color of the background you can use the **GCOLOR.R** routine, and set **COLOR.INDEX = 0** to a desired RGB combination. Here is an example:

```
Main
''Open graphics window with specified coordinates and a title,
''and background color

   Define WINDOW.ID as integer variable

'' Set background color
   call GCOLOR.R(0,0,500,0) ''dark green

   call OPENWINDOW.R given 4096, 28672, 0 , 32767,
                           " My Title - For Simgraphics Window",
                           0
                           yielding WINDOW.ID
   call SETWINDOW.R given WINDOW.ID

'' This message will keep your window opened until you click "OK"
   call MESSAGEBOX.R ("Exit", "End of the program")
end
```

You can find a detailed explanation of the graphics routines used and their arguments in Appenidx A.

Now we will learn how to display icons in the graphics window.

2.2 Display Icons in the Default Window

To display a few icons in the window, first you have to create them using graphics editor SIMDRAW and store them in a library of graphics elements. We have created three icons: **car.icn, plane.icn** and **image.icn** and preserved them in the library called **graphics.sg2.** This library is the default SIMGRAPHICS II library of graphical elements. It is searched first whenever you want to display a certain graphics element. Here is an example of how you display three icons from the default library in the default graphics window. You must declare graphical entities in the preamble using the SIMSCRIPT II.5 statement: **Dynamic graphic entities include...** and associate their graphical representation with an icon using **display entity_name with icon_name** in the program. As part of the initialization of the graphics system, the world view and view port have to be set before you draw in the window. Routine **SETWORLD.R** and variable **VXFORM.V** are provided for this purpose.

```
Preamble
'' Example "Show Icons"
   Dynamic graphic entities include IMAGE1, IMAGE2, IMAGE3
End ''Preamble

Main
''Set world view and view port
   Let VXFORM.V =1
```

```
   call SETWORLD.R( -1000.0, 1000.0, -1000.0, 1000.0)

'' Display icons from the default graphics.sg2 library
   display IMAGE1 with "car.icn"   at (   0.0 , 0.0)
   display IMAGE2 with "plane.icn" at (-500.0 , 500.0)
   display IMAGE3 with "image.icn" at ( 500.0, -500.0)

'' This message will keep the window open until you click “OK”
   call MESSAGEBOX.R ("Exit", "End of the program")
end
```

We did not have to mention the name of the library **graphics.sg2**, and we did not have to open a graphics window. This was done for us by the SIMGRAPHICS II run-time support. But, you can also create multiple libraries of graphical elements with arbitrary names and read icons from them as explained in the following paragraph.

2.3 Use of Multiple Graphics Libraries

SIMGRAPHICS II provides you with the possibility of organizing graphics elements of your application in one or multiple libraries. Graphical elements are classified in three categories: icons, graphs and forms. There is a naming convention: all icons have extension **.icn,** all graphs **.grf** and all forms have extention **.frm**. The library **graphics.sg2** will always be searched automatically, so graphical elements which are always needed should be placed in this library. Additional libraries can also be created with SIMDRAW. They must be explicitly read/searched during execution time using the **READ.GLIB.R** routine.

You can keep graphical elements from multiple applications in the common library because only elements which you want to show in your application will actually be loaded in the memory. Names of graphical elements from the same category in multiple libraries should be unique. Generally you can have two or more icons with the same name in different libraries, but the one from the last read library will take precedence.

See the following example “Window” as an illustration of the use of multiple libraries.

2.4 Example "Window"

The “Window” example given below is included with every SIMSCRIPT II.5 distribution, and summarizes what we have learned so far. It shows how to open a graphics window with a default position and size on the screen, write a title in it and shows how to display graphic entities with associated icons. In this example, library **graphics.sg2**, which is always loaded during the initialization phase of SIMGRAPHICS II, contains: **car.icn, plane.icn** and **image.icn**. Additional library **cart.sg2** contains **cart.icn**. It was read-in using library call **READ.GLIB.R** to display one of the graphical entities with **cart.icn**.

```
Preamble
'' Example "Graphics window"
   Normally mode is undefined
   Dynamic graphic    entities include IMAGE1, IMAGE2, IMAGE3, IMAGE4
```

```
End ''Preamble

Main
'' Open graphics window with specified coordinates and a title
   Define WINDOW.ID as integer variable

   call OPENWINDOW.R given 4096, 28672, 0 , 32767,
                           " My Title - For Simgraphics Window",
                           0
                           yielding WINDOW.ID
   call SETWINDOW.R given WINDOW.ID

'' Set world view and view port
   Let VXFORM.V =1
   call SETWORLD.R ( -1000.0, 1000.0, -1000.0, 1000.0)

'' Display icons from default graphics.sg2 library
   display IMAGE1 with "car.icn"   at (   0.0 , 0.0)
   display IMAGE2 with "plane.icn" at (-500.0 , 500.0)
   display IMAGE3 with "image.icn" at ( 500.0, -500.0)

'' Display icon from additional cart.sg2 library
   call READ.GLIB.R ("cart.sg2")
   display IMAGE4 with ("cart.icn") at (-500.0, -500.0)

'' This message will keep window open until you click on "OK"
   call MESSAGEBOX.R ("Exit", "End of the program")
end
```

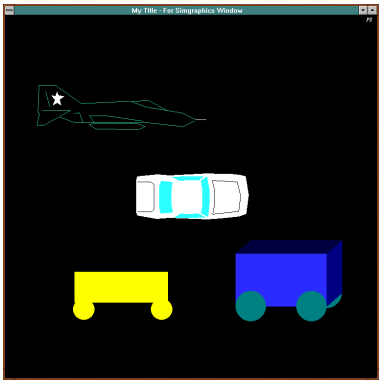


Figure 2-1. Example Window

Compile, link and execute this example. As you see, during execution time you can reposition, resize, and iconize this window. The images displayed in the window will rescale in accordance with the window dimensions.

You can experiment by changing the title, initial position, and size of the window from your program. You can also open multiple non-square windows and position them on the screen in an arbitrary way, as shown in the following example.

2.5 How to Open Multiple SIMGRAPHICS II Windows

In SIMGRAPHICS II you can programmatically open multiple non-square windows. Each window has its own title, position and world space to present. Example “nwindows” illustrates how to open three windows and display two icons in the upper window and one icon in each of the lower windows:

```
Preamble   ''Example "Multiple Graphics Windows"
     Normally mode is undefined
     Dynamic graphic entities include IMAGE1, IMAGE2, IMAGE3, IMAGE4

     Define .window1 to mean 1
     Define .window2 to mean 2
     Define .window3 to mean 3

End ''Preamble

Main
'' Open graphics windows with specified coordinates,
'' and a title

   Define WINDOW1.ID as integer variable
   Define WINDOW2.ID as integer variable
   Define WINDOW3.ID as integer variable

   call OPENWINDOW.R given 4096, 28672, 16383, 32767,
                          " Simgraphics Upper Window ",
                          0
                          yielding WINDOW1.ID

   call OPENWINDOW.R given 4096, 16383, 0, 16383,
                          " Simgraphics Lower Window1 ",
                          0
                          yielding WINDOW2.ID

   call OPENWINDOW.R given 16383, 28672, 0, 16383,
                          " Simgraphics Lower  Window2 ",
                          0
                          yielding WINDOW3.ID

   let VXFORM.V = .window1
   call SETWINDOW.R given WINDOW1.ID
   call SETWORLD.R ( -1000.0, 1000.0, 0.0, 1000.0)

   let VXFORM.V = .window2
   call SETWINDOW.R given WINDOW2.ID
   call SETWORLD.R ( -1000.0, 0.0, -1000.0, 0.0)
```

```
    let VXFORM.V = .window3
    call SETWINDOW.R given WINDOW3.ID
    call SETWORLD.R ( 0.0, 1000.0, -1000.0, 0.0)

    let VXFORM.V = .window1
    show IMAGE1 with "car.icn"   at ( 500.0 , 500.0)
    show IMAGE2 with "plane.icn" at (-500.0 , 500.0)

    let VXFORM.V = .window2
    call read.glib.r ("cart.sg2")
    show IMAGE4 with ("cart.icn") at (-500.0, -500.0)

    let VXFORM.V = .window3
    show IMAGE3 with "image.icn" at ( 500.0, -500.0)


'' This message will keep windows open until you click "OK"
    call MESSAGEBOX.R ("Exit", "End of the program")
end
```

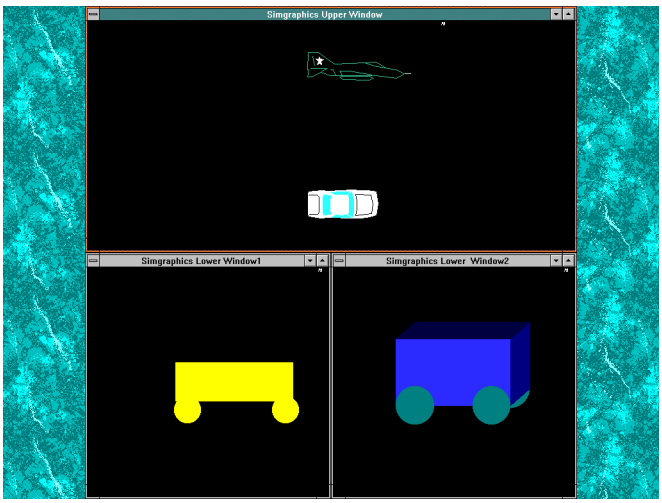


Figure 2-2. Multiple Window Example

Every window is independent and can be individually repositioned, resized and iconized. A more elaborate example, "calship," can be found in the examples of the SIMSCRIPT II.5 distribution.

Icons in the libraries **`graphics.sg2`** and **`cart.sg2`** are created without any programming using SIMDRAW. Learn how to use SIMDRAW by experimenting. First, look at the existing libraries and then draw your own graphics elements: icons, graphs, dialog boxes, menus, and palettes.

2.6 SIMDRAW — the Graphics Editor

SIMDRAW is the editor provided by the SIMGRAPHICS II system to create and edit icons or graphic images, charts and graphs and user interface items such as menu bars, dialog boxes and palettes. The editor is portable across all systems on which MODSIM III is available. It produces graphics files which are also portable.

Graphic items are built from primitives such as lines, circles, polygons, etc. Typically these parts are grouped together for convenience in handling. Each group and primitive can have a field name. This is used as a "handle" to identify the graphic objects when they are in a graphics library or when they are part of another graphics object.

Any graphics item can be saved in a graphics library.

SIMDRAW's main window is shown below. This window categorically lists all objects contained in the currently loaded library file. From this window you can create and edit images, icons, graphs (2-D charts, level meters, etc.), dialog boxes, menu bars, and palettes. A separate window is created for each object being edited. This allows you to copy parts of an object into the *clipboard* and paste them into another object of the same type. To add an object to the library, select one of the palette buttons on the left side of the window. Editing an existing object can be accomplished by double clicking on its name in the listing.

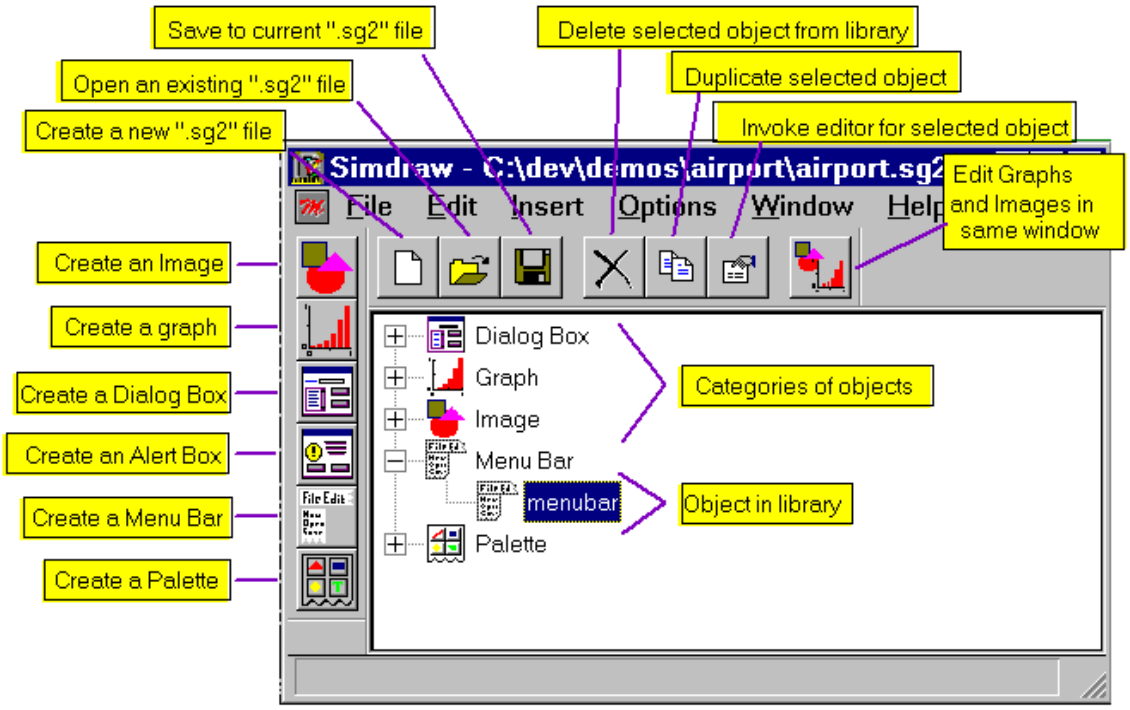
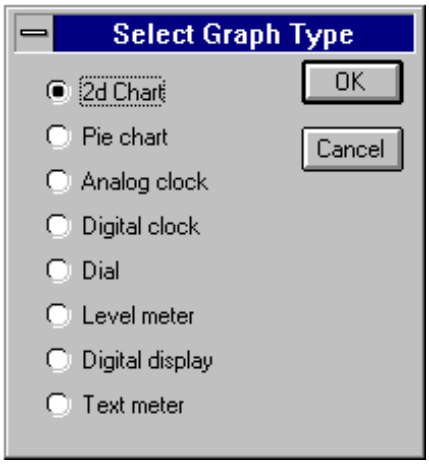


Figure 2-3. SIMDRAW Main Window

Double clicking on an image in the listing will invoke a separate window called the **Image Editor** which will contain only that image. Images are composed of circles, polygons, sectors, arcs, polylines, text, and bitmaps. Primitives are added to the image by selecting a primitive type from the **Mode** palette on the left. Bitmaps are added using the **File/Import** option. Primitives can be repositioned, resized, flipped, and rotated. The style and color of a selected primitive can be changed using the **Color** palette on the bottom and the **Style** palette on the right side of the **Editor**. Points defining a polygon or polyline can be added, removed and repositioned.

A **Layout Editor** allows you to position and resize any number of images and graphs within the same window.

The **Graph Editor** allows you to edit a variety of 2-D charts, pie charts, clocks and meters. Clicking on the **Bar Chart** palette button on the left side of the **List** window will present the following dialog:



The above graph types can be used to graphically display values of monitored or "display" variables. The **2d Chart** is used to represent SIMSCRIPT histograms. These are defined in the preamble as follows:

```
...
Define <monitored_variable> as a real variable
Tally <histogram_name>  (<lo_bound> to <hi_bound> by <delta>) as the
dynamic histogram of
<monitored_variable>
...
```

In your program, the histogram is loaded as follows:

```
...
Display histogram <monitored_variable> with "chart_name.grf"
...
```

where "`chart_name.grf`" is the *library name* given to the object in SIMDRAW.

The **Level meter, Dial,** and **Digital display** are used to represent display variables. A display varible can be defined in your preamble as:

```
...
Define <monitored_variable> as a real variable
Display variables include <monitored_variable>
...
```

and loaded into your program with a statement of the form:

```
...
Display <monitored_variable> with "meter_name.grf"
...
```

where "`meter_name.grf`" is the *library name* given to the object from within SIMDRAW.

The **Analog clock** and **Digital clock** objects display time. The preamble definition is as follows:

```
...
Define <time_variable> as a real variable
Display variables include <time_variable>
...
```

and loaded into your program with a statement of the form:

```
...
Display <time_variable> with "clock_name.grf"
...
```

where "`clock_name.grf`" is the *library name* given to the object from within SIMDRAW.

The **Pie chart** can be used to monitor an array of variables. The preamble definition is as follows:

```
...
Define <monitored_array> as a 1-dim integer array
Display variables include <monitored_array>
...
```

and loaded into your program with a statement of the form:

```
...
Display <monitored_array> with "pie_name.grf"
...
```

where "`pie_name.grf`" is the *library name* given to the object from within SIMDRAW.

The **Text meter** monitors a variable of type "text".

```
...
Define <monitored_text> as a text variable
Display variables include <monitored_text>
...
Display <monitored_text> with "text_display.grf"
...
```

2.7 Creating an Icon

One of the nice things about SIMGRAPHICS II is that changing the graphical display produced by an application program can be accomplished simply by using the editor, rather than by modifying source code. So if you would rather animate a rocket or a race car, feel free to draw one of those instead. We only picked a cart because its easy to draw—just a box with two circles for wheels!

To create the icon, you should first enter the *image editor* by clicking on the upper-left palette button on the main window. The **Image Editor** window will appear allowing you to draw lines, polygons, circles, arcs, sectors and text. You can also import bitmaps created by another drawing tool such as "MS Paint".

To create a wheel for the cart, first click on the "circle" tool on the left side palette to enter "circle drawing" mode. Now click and drag an outline of a circle in the canvas of the window. You can now resize the circle by dragging one of the small green "resize handles" of the selected circle. Moving the circle is even easier; just click and drag!

You can make the second circle by copying the first one. Use the **Edit/Copy** option to copy the selected circle into the clipboard. Now select the **Edit/Paste** option and drag the outline of the copied circle to where you want it. The body of the cart can be made with a simple rectangle. Click on the "rectangle" tool on the left palette and drag an outline for the body of your cart. You can set the color and style of these primitives by selecting them and clicking on buttons in color and style palettes on the bottom and right-hand sides of the editor window.

If you want the wheels to appear on top of the body, hold down the **<Shift>** key and click on each (unselected) circle. Both wheels should now be selected. Use the **Layout/Bring to front** option to re-stack the objects.

If you want to resize the whole image, a "group" should be made containing both wheels and the body. Click in the canvas and drag a selection rectangle over all the primitives. With both circles and the box selected, use the **Layout/Group** option to make a grouping. The resulting group can now be resized. The group can be destroyed without deleting your objects using the **Layout/Ungroup** option.

Before saving this object to the **`graphics.sg2`** file, you should define the image's properties using the **Edit/Image...** option. After the **Image Detail** dialog box appears, set the name of the object shown in the **Lib. Name** text box to **`cart.icn`**. This is the same name that is used in your program in statements of the form

```
...
```

```
Display IMAGE with "cart.icn" at (X,Y).
...
```

Next, the cart should be re-centered. Click on the **Select..** button to select a new center-point. You will then be asked to click inside the window to define the center-point. (The center point would correspond to (X,Y) in the above SIMSCRIPT code). Click on the **OK** button to return to the editor. Save the image and library file using the **File/Save** option. The edit session can be terminated by closing the editor's window. Refer to figure 2-4.

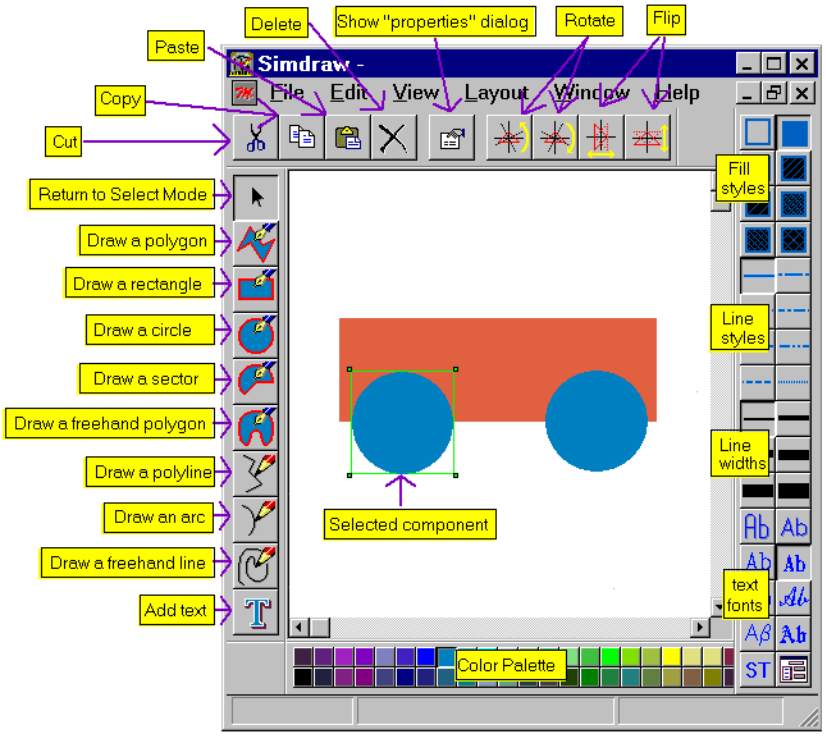


Figure 2-4. Creating a Cart Icon

2.8 Adding Animation

SIMGRAPHICS II has high expressive power. With a very few powerful graphical statements you can add animation to a graphical entity. Example "Image-1" (figure 2-5) shows a graphical entity, IMAGE, displayed with icon **`cart.icn`** in the default graphical window. Motion of the graphical entity, IMAGE, is described with the SIMSCRIPT II.5 process **`IMAGE.MOTION`** which is written using **`ORIENTATION.A`** and **`VELOCITY.A`** attributes of the graphical entity. Synchronization with real-time is achieved through the **`TIMESCALE.V`** variable. This example is included with every SIMSCRIPT II.5 distribution. It will run successfully if you added icon **`cart.icn`** to **`graphics.sg2`** using SIMDRAW in the previous paragraph. When you want to show a graphical entity with a certain icon, graphics library **`graphics.sg2`** is searched first. You can also change this example to use the library **`cart.sg2`** which contains **`cart.icn`**, or you can show

graphical entity IMAGE with some other icon like: **`car.icn`**, **`plane.icn`** or **`image.icn`**, which are in the library **`graphics.sg2`**, as shown in the example "Window," figure 2-1.

Figure 2-5. Output of the Image-1 Routine

```
Preamble   ''Example "IMAGE-1"
     Normally mode is undefined

 Define  NUM.ROTATIONS and  SPEED as double variables
   '' Animation declarations:
     Dynamic graphic entities include IMAGE
     Processes include IMAGE.MOTION

End ''Preamble

Main
'' Set up the world view and view port
         Let VXFORM.V = 1
         Call SETWORLD.R (-1000.0, 1000.0,-1000.0, 1000.0)

'' Set real-time synchronization and motion parameters
         Let TIMESCALE.V   = 100
         Let SPEED = 300
         Let NUM.ROTATIONS = 2

'' Set graphical representation for  graphics entity IMAGE,
'' Icon cart.icn is in the default library graphics.sg2.
'' Activate 'image.motion' process.
         Show IMAGE with "cart.icn"
         Activate an IMAGE.MOTION now
         Start simulation

End ''Main


Process IMAGE.MOTION
'' Describes the motion of graphical entity IMAGE
'' by setting the orientation and velocity of the IMAGE
```

```
        Define CURRENT.COURSE, NUM.SIDES, ANGLE as real variables
        Define I, J, CURRENT.TICK as integer variables

        For I = 1 to NUM.ROTATIONS
        Do
             let NUM.SIDES = 4
            Let ANGLE = (2 * PI.C * (1 - 2 * RANDI.F(0,1,2))) / NUM.SIDES

             For J = 1 to NUM.SIDES
             Do
                  Let ORIENTATION.A(IMAGE) = CURRENT.COURSE
               Let VELOCITY.A(IMAGE) = velocity.f(SPEED, CURRENT.COURSE)

                  Work (1000 * PI.C) / (NUM.SIDES * SPEED) units
                  Add ANGLE * min.f(NUM.SIDES - J,1) to CURRENT.COURSE
             Loop
        Loop
   End '' process IMAGE.MOTION
```

When declared as graphical, an entity gets a few internal attributes that are stored by the graphical system: its position, velocity, orientation, and a pointer to an icon. Some of these attributes can be accessed programmatically, either directly like **`ORIENTATION.A`** or through an attached routine/function like **`VELOCITY.A`**, which is accessed through the function **`velocity.f`** .

This example has hard coded values for icon name, and motion parameters. You can provide the user of your program with the possibility of entering these parameters in text form or with using a more convenient graphical user interface. In the following two paragraphs we will show how to create a dialog box and use it in your SIMSCRIPT II.5 program.

2.9 Creating a Dialog Box

The user of your program needs some way to influence what the program will do. To this end, SIMGRAPHICS II provides *forms*. For your simulation you can create a dialog box which allows the user to choose an icon name to represent the image entity: **`car.icn`**, **`plane.icn, image.icn`** or **`cart.icn`**. Or you could create a dialog box to define motion parameters: speed, number of rotations and time scale, and then either to signal that those values are correct by pressing an **Ok** button, or else to end the program by pressing a **Cancel** button.

To create the dialog box click on the appropriate palette button on the left-hand side palette of the main window. This action will bring up the **Dialog Box Editor** window. See figure 2-6. You can add buttons, check boxes, text boxes, etc. to a “template” dialog box shown in the **Editor** window. The template will already contain an **OK** and a **Cancel** button. You can move these buttons around by clicking and dragging them with the mouse.

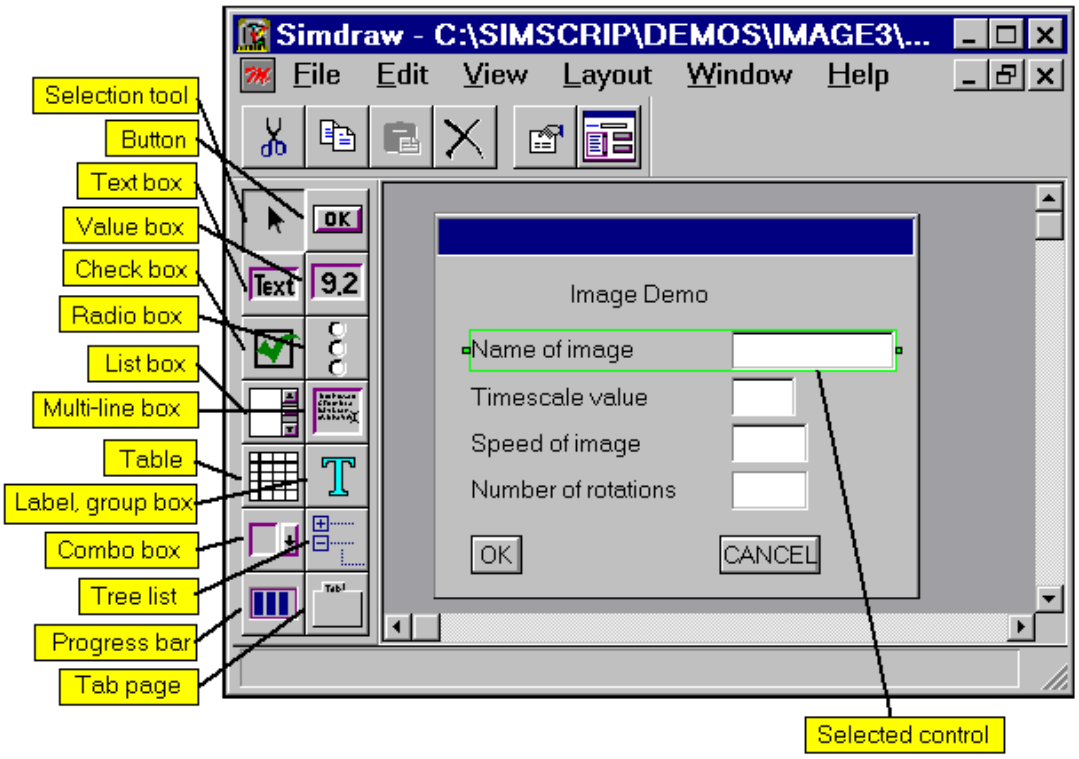


Figure 2-6. Dialog Box Editor Window

Double click on the **OK** button and a **Properties** dialog will be brought up allowing you to set its attributes. Enter **OK** in the **Field Name** text box. Bring up the **Properties** dialog for the **Cancel** buttons and enter **CANCEL** in the text box labeled **Field Name**.

Now add a text box to the dialog. Click on the "text box" button on the left palette and move an outline of the text box to your dialog. Double click on this new text box to make its **Properties** dialog appear. Enter the text "ICON NAME" into the **Field name** box. Enter the text "Name of Image" into the **Label** box. The small green boxes shown on the selected text box are called "resize handles". You can resize the text box by clicking and dragging one of these handles.

Now add three value boxes to your dialog in the same manner as above. Edit the properties of each value box and change their field names to "TIMESCALE", "SPEED", and "NUM ROTATIONS", respectively. Set the labels to "Timescale value", "Speed of image", and "Number of rotations", respectively.

This dialog is now almost completed. Before you save it, you need to assign properties to the dialog box itself. Double click on the header bar of the sample dialog to bring up its properties. Enter the text "image.frm" into the **Lib. Name** box. This same text string is used to load the dialog box into your program! Enter "Image Demo" into the **Title** box to set the

title displayed in its header bar and then return to the editor by clicking on **OK**.
If you want to see what the "real" dialog will look like, use the **Layout/Show dialog** option. Use its "go away" button in the header bar to erase it. Use the **File/Save** option to save both the dialog and "**`graphics.sg2`**".

2.10 Adding Graphical User Interaction Using Dialog Boxes

Example IMAGE-2 is the same as IMAGE-1 with only one addition. It uses a dialog box to accept user choice for icon name, speed, time scale and number of rotations. See figure 2-7. All added lines for GUI are in **main** and are highlighted.

To add a graphical user interface to example IMAGE-1, you only need to create a dialog box and to use it in your program through appropriate graphical routines and attributes like: **`accept.f, dfield.f, DTVAL.A`** and **`DDVAL.A`**.

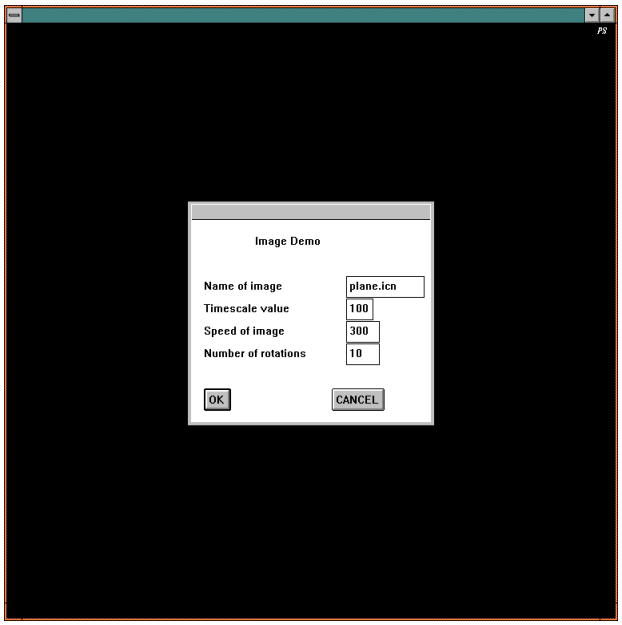


Figure 2-7. Dialog Box for Example IMAGE 2

```
Preamble   ''Example "IMAGE 2"
     Normally mode is undefind
     Define  NUM.ROTATIONS, SPEED as double variables

     '' Animation declarations:
     Processes include IMAGE.MOTION
     Dynamic graphic entities include IMAGE

End ''Preamble

Main
     '' Forms definitions:
```

```
      Define FORM.PTR as a pointer variable

      '' Set up the view port and world view
      Let VXFORM.V = 1
      Call SETWORLD.R (-1000.0, 1000.0,-1000.0, 1000.0)

      '' Display the form and accept model parameters
        Show FORM.PTR with "image.frm"
        If accept.f(FORM.PTR, 0) eq "OK"
          '' Set timescale and speed:
          '' timescale is the number of 1/100ths seconds
          '' that will pass for every simulation time unit.
          '' The speed is in real world units / second.

          Let TIMESCALE.V = DDVAL.A(dfield.f("TIMESCALE", FORM.PTR))
          Let SPEED = DDVAL.A(dfield.f("SPEED", FORM.PTR))

          Let NUM.ROTATIONS = DDVAL.A(dfield.f("NUM ROTATIONS", FORM.PTR))


          '' Accept icon name for graphics entity IMAGE,
          '' show it with that name and
          '' activate an 'image.motion' process.

           Show IMAGE with DTVAL.A(dfield.f("ICON NAME", FORM.PTR)
           Activate an IMAGE.MOTION now

           Start simulation
       Always
End ''Main


Process IMAGE.MOTION
     Define CURRENT.COURSE, NUM.SIDES, ANGLE as real variables
     Define I, J, CURRENT.TICK as integer variables

     For I = 1 to NUM.ROTATIONS
     Do
          let NUM.SIDES = 4
          Let ANGLE = (2 * PI.C * (1 - 2 * RANDI.F(0,1,2)))
               / NUM.SIDES

          For J = 1 to NUM.SIDES
          Do
               Let ORIENTATION.A(IMAGE) = CURRENT.COURSE
               Let VELOCITY.A(IMAGE) = velocity.f(SPEED,
                                              CURRENT.COURSE)

               Work (1000 * PI.C) / (NUM.SIDES * SPEED) units
               Add ANGLE * min.f(NUM.SIDES - J,1) to CURRENT.COURSE
          Loop
     Loop
End '' process IMAGE
```

To show simulation time changes is easy. We create a smart icon clock, and associate it with a display variable which will be updated every time simulation time changes. First, though, learn how to create a smart icon graph with SIMDRAW.

2.11 Creating a Graph

Run SIMDRAW, load the library and create a graph clock by clicking on the palette button on the left hand side of the main window. Choose **Analog clock** and click **OK** to bring up the **Graph Editor** and a "clock" template. Its easy to change the styles and colors of any part of the clock. Just select the part of the clock you wish to change, and use the style and color palettes.

To move the clock just click and drag with the mouse. To resize it, drag a selection box over the entire graph. This will select the *whole* clock and not just a part of it. You can now resize it using the small green resize handles on the selection box.

You can change properties of the clock by double clicking on it to bring up the **Clock Detail** dialog. Enter **`image.grf`** into the **Lib. Name** box and "Simulation time" into the **Title** box, and then click **OK**. You can now save this clock and **`graphics.sg2`** with the **File/Save** option.

2.12 Adding Presentation Graphics

Example IMAGE-3, figure 2-8, is the same as IMAGE-2, except for the addition of presentation graphics. It defines a display variable **`CLOCK.TIME`** and uses **`image.grf`** for representing simulation time. User written routine `CLOCK` is supplied to update the display variable **`CLOCK.TIME`** and to synchronize with real-time through the variable **`TIMESYNC.V`**.

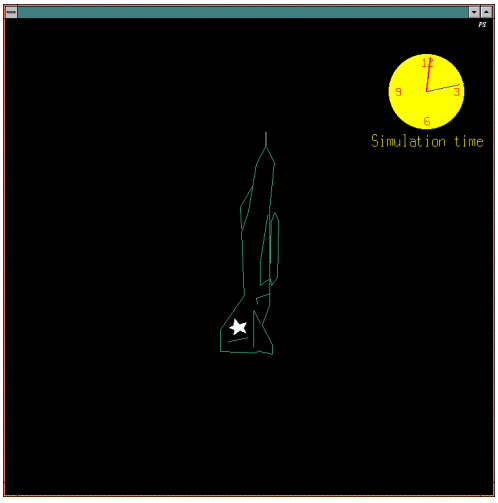


Figure 2-8. Example IMAGE-3

```
Preamble   ''Example "IMAGE-3"
     Normally mode is undefined
     Define  NUM.ROTATIONS, SPEED as double variables

'' Animation declarations:
     Processes include IMAGE.MOTION
     Dynamic graphic entities include IMAGE

'' Presentation graphics declarations:
     Define CLOCK.TIME as double variables
     Display variables include CLOCK.TIME

End ''Preamble
Main
'' Forms definitions:
     Define FORM.PTR as a pointer variable

'' Set up the world view and view port
     Let VXFORM.V = 1
     Call SETWORLD.R (-1000.0, 1000.0,-1000.0, 1000.0)

'' Presentation graphics:
     Let TIMESYNC.V = 'CLOCK'
     Show CLOCK.TIME with "image.grf"

'' Display the form and get model parameter
     Show FORM.PTR with "image.frm"
     If accept.f(FORM.PTR, 0) eq "OK"

         '' Set timescale and speed:
         '' timescale is the number of 1/100ths seconds
         '' that will pass for every simulation time unit.
         '' The speed is in real world units / second.

         Let TIMESCALE.V = DDVAL.A(dfield.f("TIMESCALE", FORM.PTR))
         Let SPEED = DDVAL.A(dfield.f("SPEED", FORM.PTR))
         Let NUM.ROTATIONS = DDVAL.A(DFIELD.F("NUM ROTATIONS",
                                              FORM.PTR))

         '' Accept icon name for graphics entity IMAGE,
         '' show it with that name and
         '' activate an 'image.motion' process.

          Activate an IMAGE.MOTION now
          Show IMAGE with DTVAL.A(dfield.f("ICON NAME", FORM.PTR))

          Start simulation
     Always
End ''Main

Process IMAGE.MOTION
     Define CURRENT.COURSE, NUM.SIDES, ANGLE as real variables
```

```
     Define I, J as integer variables

     For I = 1 to NUM.ROTATIONS
     Do
          Let NUM.SIDES = 4
          Let ANGLE = (2 * PI.C * (1 - 2 * RANDI.F(0,1,2))) / NUM.SIDES

          For J = 1 to NUM.SIDES
          Do
               Let ORIENTATION.A(IMAGE) = CURRENT.COURSE
               Let VELOCITY.A(IMAGE) = velocity.f(SPEED,
                                                  CURRENT.COURSE)

               Wait (1000 * PI.C) / (NUM.SIDES * SPEED) units
               Add ANGLE * min.f(NUM.SIDES - J,1) to CURRENT.COURSE
          Loop
     Loop
End '' process IMAGE

Routine CLOCK given TIME yielding NEWTIME
     Define  TIME, NEWTIME as double variables

     Let CLOCK.TIME = TIME / (24*60*60)
     Let NEWTIME = TIME
return
end
```

This is a simple, yet complete, example with all three types of SIMGRAPHICS elements represented: animation graphics, input forms and presentation graphics.

2.13 Creating a PostScript File

SIMGRAPHICS II provides an automatic way of creating PostScript files for better documentation of a simulation process. Without additional programming, you can automatically create snap shots in PostScript form during a simulation run. Execute any of the tutorial examples and click on the PostScript icon (the small “PS” inside a circle) in the upper-right corner of the SIMGRAPHICS II window. A PostScript file **`print1.ps`** will be created. Subsequent clicks on the icon will create a snapshot named **`print2.ps`**, etc. These encapsulated PostScript files can be printed on a PostScript compatible laser printer or imported into a text processor for improved documentation of the simulation progress and simulation results.

2.14 Using a Bitmap as a Background

Many simulation models require use of geographical maps, road maps or airport layouts as a background, with airplanes, vehicles or other icons moving in the foreground. Maps or other color images can be transformed to raster files or bitmap files using a color scanner. Acceptable file formats for bitmaps are Windows Bitmap (`.BMP`) on PCs and X-Windows format (`.xwd`) on UNIX platforms.

Bitmaps can be used as icons, as static objects in a background, or as dynamic moving super-imposed objects.

First, create raster/bitmap images using any color scanner, or using some of the system software tools like Paintbrush on the PC Windows platform. The bitmap file you create must be in the same directory as your graphics library file. Invoke SIMDRAW, load your graphics library, and then invoke the **Image Editor** by clicking on its palette button in the top left side of the main window. From within the **Image Editor**, use the **File/Import** option and enter the name of your bitmap file. After a few seconds the bitmap should appear in your window. This bitmap can now be saved into your image object. Remember to use the **Edit/Image...** option before saving and assign the library name and a center point to your image.

In your SIMSCRIPT II.5 program, after the initialization phase of the graphics system, show your background map and then display and animate your superimposed icons. SIMGRAPHICS II will take care of the redrawing of the necessary parts of the background icon as the icons move across the screen.

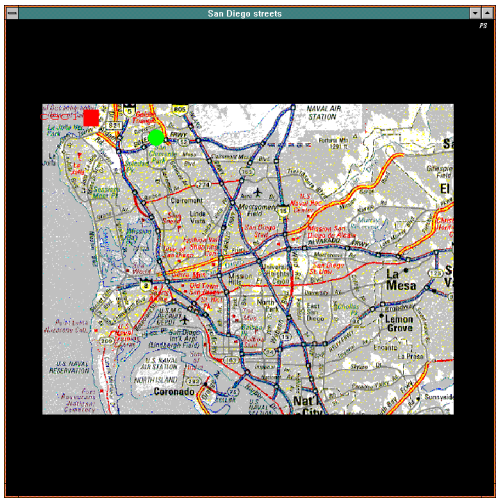


Figure 2-9. Example "San Diego" Showing Imported Bitmap

Example: "sandiego" in the SIMSCRIPT II.5 distribution, shows a San Diego road map saved as a bitmap and used as a background, with a vehicle token icon moving along the freeway.

```
Preamble   ''Example "San Diego"
     Normally mode is undefined
     Dynamic graphic entities include CITY.MAP,
         COMPANY.LOCATION,VEHICLE1
     Processes include vehicle.motion
End ''Preamble

main
'' Open graphics window with specified coordinates and a title
     Define WINDOW.ID as integer variable
     call OPENWINDOW.R given 4096, 28672, 0 , 32767,
```

```
                          " San Diego roads",  0
                          yielding WINDOW.ID
     call SETWINDOW.R given WINDOW.ID

'' Set world view and view port
     Let  VXFORM.V =1
     call SETWORLD.R ( -32767.0, 32767.0, -32767.0, 32767.0)

'' Display icons from default graphics.sg2 library
     show CITY.MAP with "sandiego.icn" at ( 0.0, 0.0)
     show COMPANY.LOCATION with "caci.icn" at (-27000.00, 11000.00)
     show VEHICLE1 with   "token.icn"   at (-4500.0, 0.0)

     Let TIMESCALE.V = 100
     Activate a VEHICLE.MOTION  now
     Start Simulation

     call MESSAGEBOX.R ("Exit", "End of the program")
end

Process VEHICLE.MOTION
   Let MOTION.A(VEHICLE1) = 'LINEAR.R'
   Let VELOCITY.A(VEHICLE1) = VELOCITY.F(700.0, PI.C/2.4)
   Work 12 units

   Let VELOCITY.A(VEHICLE1) = VELOCITY.F(700.0,(PI.C/2.4+PI.C/4.0))
   Work 14 units

   Let VELOCITY.A(VEHICLE1) = VELOCITY.F(700.0,
                                      PI.C*(1.0/2.4+1.0/4.0+1.0/3.0))
   Work 8 units

   Let VELOCITY.A(VEHICLE1) = 0
   Work 5 units
end
```

The following example, "eagle," included with the SIMSCRIPT II.5 distribution, shows how you can use a bitmap to create a realistic background for your simulation.

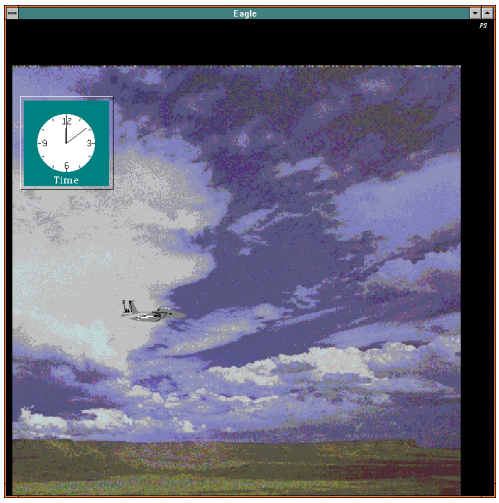


Figure 2-10. Example of a Bitmap Used as a Background

Note: Bitmap files are not copied into the library of graphical elements. Library **.sg2** only contains the names of the **.bmp** files, so if you transfer your SIMGRAPHICS II application to another directory or another system, you must transfer corresponding **.sg2** libraries and **.bmp** or **.xwd** files.

2.15 Creating Cascadeable Menus

A cascadeable menu bar is a menu bar with nested menus. In other words, it is a menu which contains other menus. An example of a simple non-nested menu bar would be a menu bar with two menus, **QUIT** and **SHOW**, where **QUIT** has two menu items **CANCEL** and **OK**, while **SHOW** has a list of countries created as menu items, for instance: USA, CANADA, JAPAN, ENGLAND and ITALY. This list can be very long, and in order to provide faster access and more structured organization we would like to introduce three menus for continents and restructure the **SHOW** menu in the following way:

```
QUIT>        SHOW>
    CANCEL       AMERICA>
    OK                      CANADA
                            USA
                   ASIA>
                            JAPAN
                   EUROPE>
                            ENGLAND
                            ITALY
```

This is an example of cascadeable menus. To create a cascadeable menu bar is easy. You use SIMDRAW as with any other graphical element and store it in a library **.sg2**. Example **menus** included with every SIMSCRIPT II.5 distribution has a library of graphical elements **menu.sg2** with **showmenu.frm** which represent a cascacadeable menu as a form.

If you want to see this form, start SIMDRAW, load **menu.sg2**, and then double click on the object **showmenu.frm**. This will invoke the **Menu Bar Editor**. A template of the **showmenu.frm** menu bar is displayed in the window. You can interact with this menu bar by clicking on its component menus. Unlike a "real" menu bar, many menus can be pulled down at once. This allows you to transfer submenus and menu items from one place to another on the menu bar.

Double-clicking on any menu or menu item label will invoke its **Properties** dialog. This allows you to see the **Field name** used by your program to reference the sub-menu or item. You can add sub-menus and menu items with the **sub-menu** and **menu item** palette buttons on the left hand side of the window. Just drag a new menu or item from the palette to the appropriate place on the template. Use the **Layout/Show menu bar** option to see the "real" menu bar.

Before saving a menu bar, remember to double-click on the bar itself and set its **Lib. Name** field to **showmenu.frm**.

2.16 Using Cascadeable Menus

In this section we will show how to use the created cascadeable menu bar in a SIMSCRIPT II.5 program.

From SIMDRAW you can define one of the following three actions for **accept.f** to take when displaying your menu bar:

Asynchronous If a simulation is running, suspend the active process. Reactivate this process when the control routine returns a STATUS value of "1".

Synchronous Wait in **accept.f** until the control routine returns a STATUS value of "1". Useful for programs not involving simulation.

Don't Wait **accept.f** will display the menu bar and return immediately. The control routine will be called whenever a menu item is selected. Useful if other dialog boxes and palettes are to be displayed simultaneously.

2.16.1 Cascadeable Menus in Simulation Programs

We will first concentrate on the asynchronous technique which facilitates asynchronous user interaction during a simulation run. It automatically suspends the current simulation process and transfers control to the menu bar process whenever the user clicks on the menu bar. Graphical function **accept.f** is used to detect and accept user input and transfer control to the control routine for the menu bar. The menu bar control routine is application-specific and must be written by the program implementor. The reference name of the last

selected menu item is passed to the provided menu bar control routine, which contains a case statement with the defined actions for each menu item's reference name. When a required action is finished, the menu bar process is suspended and control is transferred back to the suspended simulation process.

In figure 2-11, we have used an asynchronous cascadeable menu bar represented with the **showmenu.frm** from the library **menu.sg2**. The basic action in this example is to show the map of a country selected from the cascadeable menus and to exit the program using **QUIT** or **OK** from the menu. Actions for every menu item in cascadeable menus are specified in the routine **MENU.CTRL**. You do not have to provide actions for intermediate menus because they are not selected. Only menu items are selected. Navigation through menu structures is automatically performed by SIMGRAPHICS II run-time support. Routine **MENU.CTRL** is passed as argument to **accept.f** in the process **DISPLAY.MENUBAR** activated from main, before the start of the simulation. Another process **COUNTER** is created here only to illustrate asynchronous transfer of control from the menu bar to simulation processes and vice versa. This process will open a text window and count and print numbers whenever it is active. Click on the graphics window to bring it up-front and use the cascadeable menu bar.

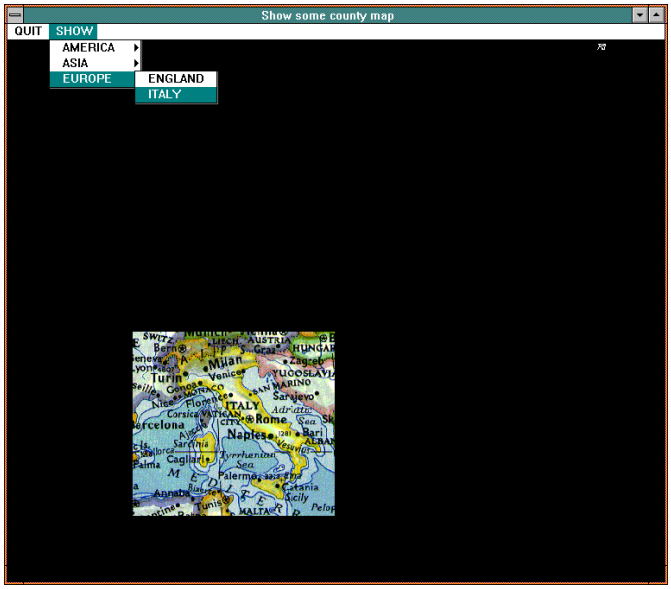


Figure 2-11. Cascadeable Menu

```
Preamble
''*************************************************************
''* Example: Asynchronous cascadeable menubar                 *
''*          Show the map  of a  country                      *
''*                                                           *
''* Structure of the menubar represented with showmenu.frm is: *
''*                                                           *
''* QUIT>             SHOW>                                   *
''*      OK               AMERICA>                           *
''*      CANCEL                   CANADA                     *
''*                               USA                        *
''*                       EUROPE>                            *
''*                               ENGLAND                    *
''*                               ITALY                      *
''*                       ASIA>                              *
''*                               JAPAN                      *
''*                                                          *
''*************************************************************
   Normally mode is undefined
   Processes include DISPLAY.MENUBAR, COUNTER
   Dynamic graphic entities include USA.MAP, ITALY.MAP, ENGLAND.MAP,
                                    CANADA.MAP, JAPAN.MAP
   Define QUIT.OK as integer variable
End

Main
   Call INIT.GRAPHICS
   Activate a DISPLAY.MENUBAR now
   Activate a counter now
   Start simulation
   Call MESSAGEBOX.R("Exit", "End of this demo")
End

Routine INIT.GRAPHICS
   Define WINDOW.ID as integer variable  '' non-square window
   Define .XLO to mean  4096             ''   o-----o xhi,yhi
   Define .XHI to mean  32786-4096       ''   |     |
   Define .YLO to mean  4096             ''   o-----o
   Define .YHI to mean  32786           '' xlo,ylo

   Call OPENWINDOW.R (.XLO, .XHI, .YLO, .YHI, "Show some county map", 0)
                      yielding WINDOW.ID
   Call SETWINDOW.R(WINDOW.ID)
   Call READ.GLIB.R("menu.sg2")
End

Process DISPLAY.MENUBAR
''This process will always be activated whenever we click on menu bar
''It will be distroyed and menubar will disappear when MENU.CTRL
''routine returns Status = 1,in our example when we click on QUIT-OK

   Define FIELD.NAME as text variables
   Define MENU.PTR as pointer variables
```

```
   Display MENU.PTR with "showmenu.frm"
   FIELD.NAME = accept.f(MENU.PTR,'MENU.CTRL')
End

Routine MENU.CTRL given FIELD.ID, FORM yielding STATUS
   Define FIELD.ID as text variables
   Define FORM as pointer variables
   Define STATUS as integer variables

   Select case FIELD.ID
      case "OK"
         let QUIT.OK = 1
         STATUS = 1'' exit from accept.f
      case "CANCEL"
      case "ENGLAND"
         erase USA.MAP, ITALY.MAP, CANADA.MAP, JAPAN.MAP
         show  ENGLAND.MAP with "england.icn" at (4096.0, 26000.0)
      case "ITALY"
         erase USA.MAP, ENGLAND.MAP, CANADA.MAP, JAPAN.MAP
         show  ITALY.MAP with "italy.icn" at (4096.0, 4096.0)
      case "USA"
         erase ENGLAND.MAP, ITALY.MAP,CANADA.MAP, JAPAN.MAP
         show  USA.MAP with "usa.icn" at (4096.0, 4096.0)
      case "CANADA"
         erase USA.MAP, ENGLAND.MAP,ITALY.MAP, JAPAN.MAP
         show  CANADA.MAP with "canada.icn" at (10000.0, 16000.0)
      case "JAPAN"
         erase USA.MAP, ENGLAND.MAP,ITALY.MAP, CANADA.MAP
         show  JAPAN.MAP with "japan.icn" at (16000.0, 16000.0)
      case "INITIALIZE"
      case "BACKGROUND"
      default
   Endselect
End

Process COUNTER
''This process is here only to ilustrate asynchronous menus
'' It will be suspended whenever we click on menu bar
  While QUIT.OK <> 1
  do
     COUNT = COUNT + 1.0
     Wait 5 units
     Print 1 line with COUNT thus
     count = **********
  Loop
End
```

3. SIMDRAW

3.1 SIMDRAW Overview

SIMDRAW is an interactive menu based program for creating and editing SIMGRAPHICS II objects. These objects can be used for animation, presentation graphics, and interactive graphical input. Types of objects include *images*, *dialog boxes*, *menu bars*, *palettes*, and various charts and graphs. These objects are saved to and loaded from SIMGRAPHICS II **.sg2** files that can be accessed by a SIMSCRIPT II.5 program.

Animation graphics or *images* are built by drawing lines, circles, polygons, arcs, sectors, bitmaps, and text. These primitives can be grouped together to form more complex images containing parts that can be manipulated independently by the application program. Images are built by the **Image Editor**.

Presentation graphs are constructed by setting attributes such as titles, minimums, maximums, etc. Several different graph types can be built. They include 2-D plots, level meters, pie charts, trace plots, clocks, dials, text displays, and digital displays. All graph types are built with the **Graph Editor**.

A **Layout Editor** is available for sizing and positioning multiple graphs and images within the same window.

Using the **Dialog Editor**, dialog boxes can be constructed for receiving interactive modal or modeless data input. The dialog box can contain buttons, check boxes, text boxes, combo boxes, list boxes, and radio button fields. A dialog box can also contain the more complicated multi-line text boxes and 2-D tables. Tabbed dialog boxes can be created.

Menu bars can be built with the **Menu Bar Editor** for receiving modeless command input. Menus can be attached to other menus producing any desired level of depth. Menu option keyboard accelerators and mnemonic keys can be defined.

Palettes are built with the **Palette Editor** for receiving simple command input. They can be (initially) docked on any edge of the window or can be floating. A palette contains palette buttons and separators.

3.2 Running SIMDRAW

SIMDRAW can be started from within SIMSCRIPT II.5, or from the command line. Upon execution a main window containing a palette and toolbar is displayed (figure 3-1). The window will contain a listing of the currently loaded SIMGRAPHICS II library. The palette on the left is used to add new objects to the library.

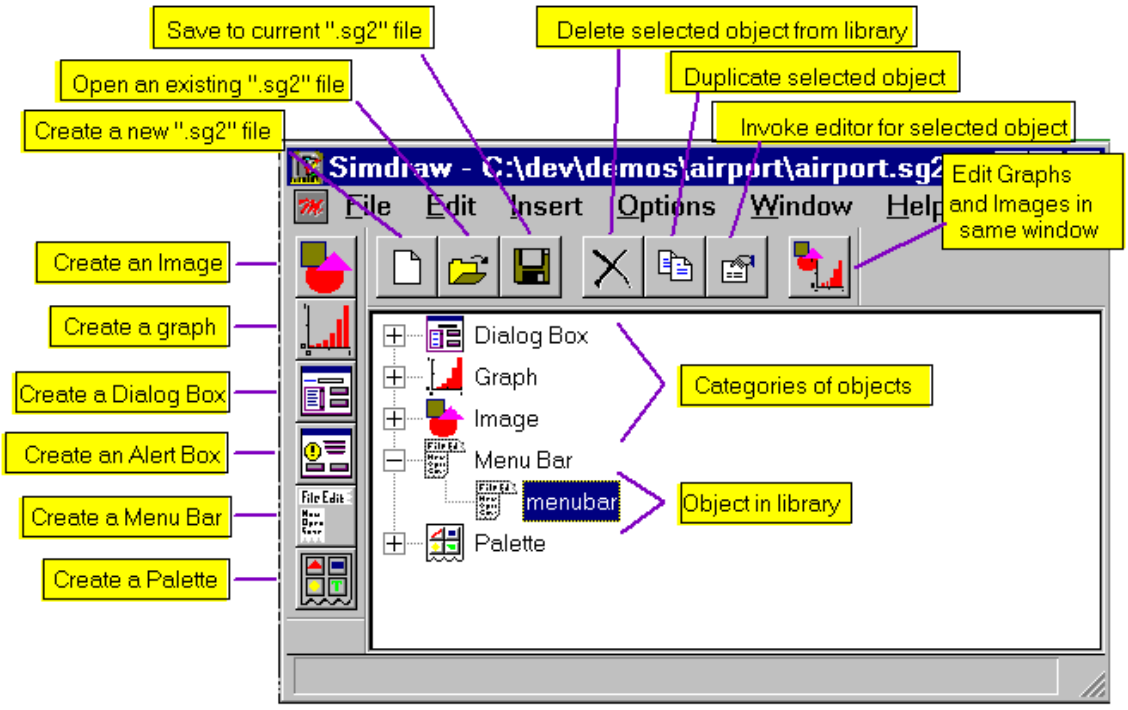


Figure 3-1. Main Window

3.3 Loading and Saving SIMGRAPHICS II Files

The **File/Open...** menu option will load an existing SIMGRAPHICS II library file and show its objects in the list window. Use the **File/Save** or **File/Save As** menu option to save all objects shown in the list window, including objects being edited. Use the **Options/Binary File** menu option to toggle between saving the file in ASCII or binary format.

3.4 Editing an Existing Object

To edit one of these objects, select its name in the listing, and then use the **Edit/Properties** menu option or the **Properties** toolbar option. A new window containing the appropriate editor will appear showing its graphical representation. After moving, resizing, or changing attributes of the object and its sub-components, select the **File/Save** or **File/Save As** menu option to write this object to its SIMGRAPHICS II library file. To end editing of this object, close its editor's window using the "go away" button in the top left corner of the window's header bar.

3.5 Adding an Object to the Library

Objects can be added to this library file by clicking on one of the "create" buttons on the left palette, or by using the **File/Insert** menu option. Creating an object will automatically invoke the editor for that object.

3.6 Removing an Object from the Library

To remove an unwanted object from the current library, select the object's name in the listing, and then use the **Edit/Clear** menu option. The library must be saved using **File/Save** before this change is permanent.

3.7 Making a Duplicate of an Object

Any graphical object in the library can be duplicated by selecting its name in the main list and then using the **Edit/Duplicate** menu option. The library must be saved using **File/Save** before this change is permanent.

3.8 Changing the Name of an Object

To change the name of an object shown in the main list, select it and use the **Edit/Properties** menu option to bring up its editor. Use the **Edit/Properties** menu option of this editor to obtain a dialog box showing the object's attributes. Change the **Library Name** text field to the new name, and then save the object with the **File/Save** menu option.

3.9 Adding an Object from Another Library

If you want to add object(s) contained in a different SIMGRAPHICS II file, use the **File/Merge...** menu option. Once a file is selected, a list box containing the names of all objects in this source library will be displayed. Choose the objects you wish to copy to your library. The **Shift** and **Ctrl** keys can be used in conjunction with the mouse to select multiple objects.

3.10 Editing Images and Graphs in Same Window

Sometimes a set of images and/or graphs must be displayed in the same context to get their size and position correct. Multiple objects can be positioned and resized from within one window using the **Layout Editor**. Select the **Layout** button on the far right-hand side of the toolbar. Using the **Shift** and **Ctrl** keys, select the set of images and graphs to be resized and positioned from the list box. After editing the objects, use the **File/Save** menu option to save all edited objects to the SIMGRAPHICS II file.

3.11 User Preferences

You can set preferences regarding the order in which objects in the currently loaded library are listed using the **Options/Preferences** menu option. One of the following three methods can be used to list your objects:

- **Time of creation** — Objects are ordered based on time of creation. The last objects added to the library are show at the bottom of the list.
- **Alphabetical** — Alphabetical order based on name
- **Categorical** — Objects are listed categorically. The categories are: **Image, Graph, Dialog Box, Menu Bar**, and **Palette**. A "heading" is created for each category, with the objects listed alphabetically under the appropriate heading. Chick on the (**-**) to the left of the heading to expose the names of the objects. To collapse the list, click on the (**+**).

The **Preferences** dialog also has options for specifying how SIMDRAW is started up. SIMDRAW can be configured to "remember" its previous window position, and which library file was loaded. Objects being edited in either the **Layout Editor**, or a **Single Editor** can optionally be reloaded at startup.

3.12 Command Line Arguments

SYNOPSIS:

```
simdraw[-l file_name] [-S sys_path_name]
 [-B sys_path_name] [-sim] [-e] [-dim xlo ylo xhi yhi]
 [-nodialog] [-noimage] [-nograph] [-nomenu] [-nopalette]
 [-W path_name]  [object names]
```

The following command line arguments are recognized by SIMDRAW:

`-l file_name`	Specifies the name of the SIMGRAPHICS II graphics file to edit.
`-e, -single`	Specifies "single edit" mode. The specified objects will be edited with no control window containing library information.

`-nodialog`
`-nograph`
`-noimage`
`-nomenu`
`-nopalette`
	Eliminates editing of the specified object types.

`-dim xlo ylo xhi ylo`	Specifies the default real world coordinate space used by the **Image Editor**.
`-B path_name`	Specifies the path to bitmap files used by SIMDRAW.
`-S path_name`	Specifies the path to system files needed to run SIMDRAW (trailing delimiter '/' or '\' must be included).
`-W path_name`	Specifies path to user SIMGRAPHICS II files.

3.13 Using the Image Editor

The **Image Editor** is used to create and edit *primitives* such as lines, polygons, circular objects, and bitmaps. Primitives can be grouped hierarchically into *images*. The editor window contains three palettes: **Mode, Style,** and **Color**. The **Mode** palette on the left side of the window is used for adding primitives.

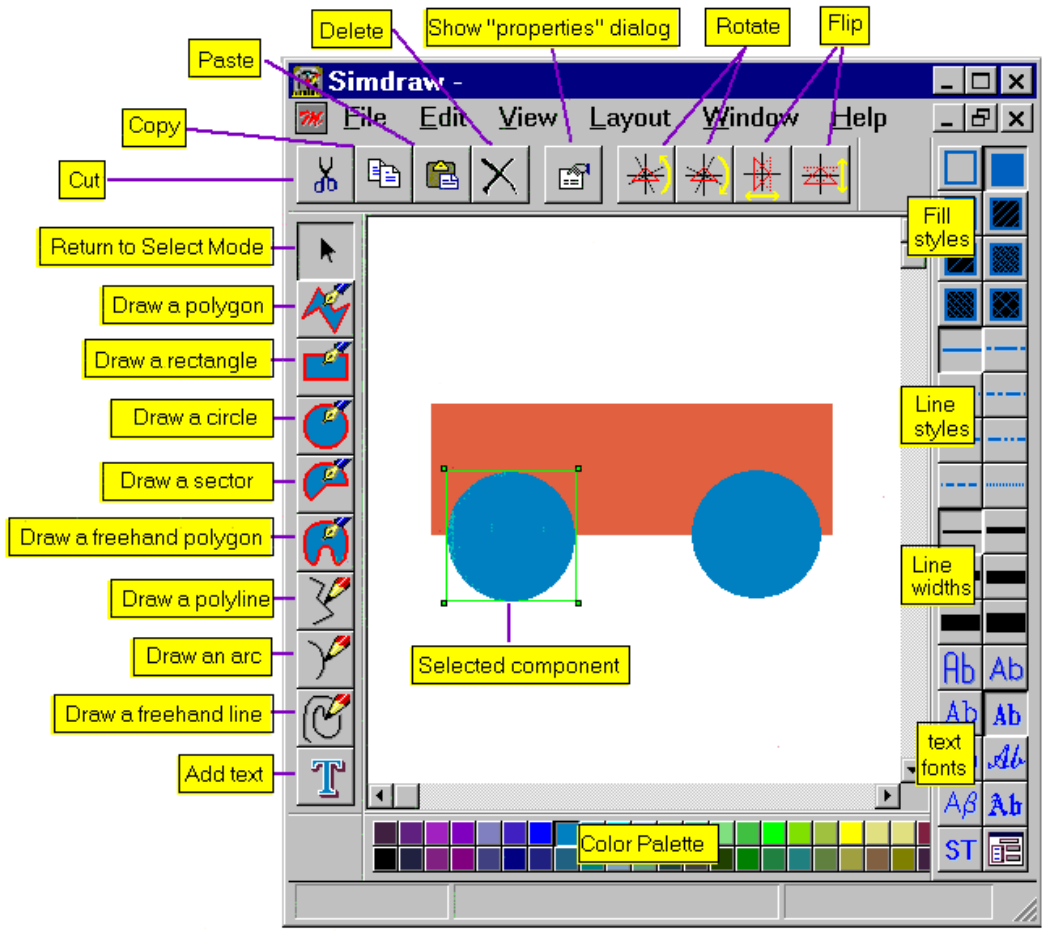


Figure 3-2. Image Editor

3.13.1 Mode, Style, and Color Palettes

The **Style** palette contains the set of dash styles, hatch styles, line widths, and text fonts that can be applied to the primitives. The **Color** palette contains 64 colors that can also be applied to the primitives. When a primitive is selected, the **Style** and **Color** palettes will be updated to reflect the style and color of that primitive. At this time, **Style** and **Color** palette changes will also be applied to the selected primitive.

The **Mode** palette is shown on the left-hand side of the **Image Editor** window. Use it to add primitives to your drawing. Refer to paragraph 3.13.4.

3.13.2 Selecting, Moving, and Resizing

Shapes are selected by clicking the mouse button over the desired shape. For example, polylines must be selected by clicking on the line itself, NOT in the line's bounding box. Multiple shapes are selected by holding down the **Shift** key and clicking on several shapes. Multiple shapes may also be selected by clicking in the background of the window and dragging the mouse over the shapes you want to select.

A grouping of shapes is selected by clicking on one of the objects in the group. Subsequent clicks over the group will select shapes within that group. Primitives inside a group can be selected directly by holding down the **Ctrl** key and clicking on the shape. Using the **Ctrl** key, subsequent clicks will select the groups *containing* the currently selected shape.

Selected shapes are marked by a bordering green or cyan box. Sides and corners of this box contain eight small square resize handles. Resizing is performed by clicking down and dragging a resize handle.

Click and drag a shape to move it to the desired position. Be careful not to click down on the resize or point handles.

3.13.3 Using the Clipboard (Cut, Copy, Paste Commands)

The **Image Editor** supports the standard cut, copy, and paste operations found under the **Edit** menu. The **Cut** option deletes selected shapes and places them in the clipboard. The deleted item remains on the clipboard until the next time a **Cut** or **Copy** is performed. You can use the **Paste** option to paste as many copies as you want from the clipboard into the image. Shapes can be deleted without changing the clipboard by using the **Delete** option.

The clipboard is shared among all active **Image Editor** sessions. You can copy graphics from one image into another by activating the source edit window, using the **Copy** option, activating the destination editor and using the **Paste** option.

3.13.4 Importing / Exporting to Other Graphical Formats

Using SIMDRAW, you can import graphics created by other graphics editors. This is accomplished by invoking the **Image Editor** and using the **File/Import** option. Graphics files in any of the following formats can be loaded into the editor:

- MS Windows Bitmaps (**.bmp**) (MS Windows only). Note that the bitmap file must reside in the same folder as your **.sg2** file.
- X Window Dump (**.xwd**) (X-Windows only). The raster file must reside in the same directory as your **.sg2** file.
- AutoCAD files (**.dxf**). Simple 2d AutoCAD files can be imported. The vector description will be maintained.
- Windows Metafile (**.wmf**) (MS Windows only). The vector description will be maintained.

You can also convert an existing SIMGRAPHICS II drawing into one of the following formats through the **File/Export** option:

- MS Windows Bitmaps (**.bmp**) (MS Windows only).
- X Window Dump (**.xwd**) (X-Windows only)
- EPS Color PostScript (**.eps, .ps**)

When exporting to **.bmp** or **.xwd** files, a mask bitmap will automatically be created. The mask file is needed to maintain transparency when rendering non-rectangular bitmaps. The mask file will be named after the export file but will have an "**m**" character appended to the file-name. (Exporting to the file **test.bmp** will automatically create **testm.bmp**.) SIMGRAPHICS II will automatically try to load the mask file whenever the original bitmap is loaded. The mask can be deleted if it is not needed.

3.13.5 Creating Primitives

The **Image Editor** supports creating and editing seven different primitive types. The primitives are polygons, polylines, circles, arcs, sectors, text, and bitmaps.

Polylines

Polylines are created by clicking either the freehand or polyline buttons on the **Mode** palette. To create a polyline, select the polyline button on the mode palette. Point to where you want to start the line and drag to draw a line segment. Continue pointing and clicking until all but the last line segment has been defined. Double click to create the last vertex and return to **Select** mode.

To create a freehand polyline press the freehand line button on the **Mode** palette. Drag the mouse around the canvas area to draw the line. Releasing the mouse button will return you to **Select** mode.

Use the **Style** palette to define dash style and line width. There are eight dash styles and six line widths to choose from.

Another attribute of the polyline is *rounding*. Corners defined by intersecting line segments can be given a rounded edge by selecting the polyline, and using the **Edit/Properties...** menu option. The **Round Corners By** value box contains the length of the segment adjacent to each vertex to be replaced by a rounded corner. This value is specified with respect to the real world coordinate space or *dimension* of the editor (the default dimension is [0, 0, 32767, 32767]). A value of 1000.0 is reasonable for rounding corners.

Polygons

Polygons are created by clicking either the *freehand*, *polygon*, or *rectangle* buttons on the **Mode** palette. To create a polygon, press the **Polygon** button on the **Mode** palette. Point and click in the window to define vertices. Double click to create the last vertex and return to **Select** mode.

To create a freehand polygon press the **Freehand** fill button on the **Mode** palette. Drag the mouse around the canvas area to draw the shape. Release the mouse button to return to **Select** mode.

To create a simple rectangle press the **Rectangle** button on the **Mode** palette. Point to where you want the lower left-hand corner of the rectangle to start, and drag the mouse to the desired top right corner. Release the mouse button to return to **Select** mode.

Use the **Style** palette to define a hatch pattern. There are eight patterns to choose from.

Circles

Circles are added by pressing the **Circle** button on the **Mode** palette. In **Circle** mode, point to where you want the center of the circle to go and drag the mouse to define the radius. Release the mouse button to draw the circle and return to the **Select** mode.

Use the **Style** palette to give the circle a hatch pattern. There are eight patterns to choose from.

Sectors

A sector is a filled semicircular shape similar to a pie slice. Sectors are composed of a center point, a starting point and an ending point, and are drawn counterclockwise from the starting point to the ending point. To draw a sector, first press the **Sector** button on the **Mode** palette. Point to where you want the center point of the sector to go, and drag the mouse.

Release the mouse over where you want the starting point of the arc to go. Drag the mouse to where you want the sector to end and release to return to **Select** mode.

Use the **Style** palette to give the sector a hatch pattern. There are eight patterns to choose from.

Arcs

An arc is a curved line contained on the circumference of a circle. Arcs are composed by a center point, a starting point and an ending point, and are drawn counterclockwise from the starting point to the ending point. To draw an arc, first press the **Arc** button on the **Mode** palette. Point to where you want the center point of the arc, and drag the mouse. Release the mouse over where you want the starting point of the arc. Drag the mouse to where you want the arc to end and release to return to **Select** mode.

Use the **Style** palette to define dash style and line width. There are eight dash styles and six line widths to choose from.

Text

Single line text primitives can be created and added to your image. To create a text primitive, press the **Text** button on the **Mode** palette. Point to where you want the center of the text to go and click the mouse button. Use the **Edit/Properties...** menu option to define the text string to be displayed. The text string can contain more than one line.

There are two different types of text, *vector text* and *system text*. Vector text fonts are fully scaleable in any dimension and are portable between MS Windows and X Windows platforms. A vector text font can be assigned to a primitive by pressing any of the eight **Style** palette buttons showing **Ab**.

System text fonts are "built-in" to the tool kit on which your server is running. Text defined using a system font is non-scaleable and can only be resized by changing the font. A system font is defined by font name, point size, and whether or not its uses italic and/or boldface calligraphy. To assign a system font to a text primitive select the primitive, and then press the **Dialog Box** button on the lower right-hand corner of the **Style** palette. The resulting **Font** box will display all fonts, point sizes, and calligraphy styles loaded on your server. The font you select will be applied to the selected text primitive. This same font can now be applied to other primitives using the **ST** button at the lower left corner of the **Style** palette.

Text alignment with respect to the image can also be defined. For example, if you wanted a text primitive defined with a system font to remain centered as an image is scaled, its alignment should be centered horizontally and vertically using the **Edit/Properties...** menu option.

Through the **Properties** dialog box, you can define whether the text can be defined programmatically through the **DTVAL.A** attribute of its display entity field. Text color can be defined through **DCOLOR.A.** For example, if this option is chosen and the text primitive's reference name is **"MY.TEXT"**, you can include it into your program code:

```
Let DTVAL.A(DFIELD.F("MY.SHAPE", ICON.PTR)) = "Hello World"
```

Bitmaps

Bitmaps (or "snap shots") are not created directly by the **Image Editor**, but can be created using another drawing tool and can then be *imported*. On MS Windows systems, "Windows bitmap" files with the **.bmp** extension can be imported and added to your image. On X Windows systems, X Windows dump file formats ending in **.xwd** can be imported.

To add a raster file to your image use the **File/Import...** menu option. Select a **.bmp** or **.xwd** file from the dialog box and press the **OK** button to import the bitmap.

Once in the **Image Editor,** bitmaps can be resizeable or non-resizeable. To change the scalability, select the bitmap and use the **Edit/Properties...** menu option. Remember that resizing bitmaps may take longer to render the first time, and can loose meaningful pictorial information if made smaller.

Alignment can be applied to bitmaps as well as text primitives. For example, if you wanted a non-scaleable bitmap to remain centered as an image is scaled, its alignment should be centered horizontally and vertically from the **Properties** dialog.

3.13.6 Creating Images

An image represents a grouping of primitives and/or other images. Images can contain other images forming a hierarchy. To create an image, select the shapes to be grouped using the **Shift** key, and then select the **Layout/Group** menu option. The resulting group will be shown bounded by the green selection box. Use the **Layout/Ungroup** menu option to destroy an image.

An image is selected by clicking on one of the primitives within it. Repeated selections of an image will select the shapes within it. Select primitives directly by clicking on them while holding down the **Ctrl** key.

Shapes can be removed from an image by selecting the shape and using the **Layout/Remove from Group** menu option. You can also add shapes to an existing image by selecting first the shapes, then an image, and then using the **Layout/Add to Group** menu option.

3.13.7 Editing the Root Image

The editor's window shows all objects contained by the image being edited or shows the *root* image. To change properties of this image (such as its name), use the **Edit/Image** menu option to display the **Image Detail** dialog.

To reset the center point of the root image, first click on the **Select** button in the **Properties** dialog. Next, position the mouse in the canvas of the edit window over where you want the center point and click. The center can also be defined by editing the fields directly in the **Properties** dialog.

The **Image Detail** dialog can be used to specify the size and angle of rotation of the image by editing the **Width, Height** and **Rotate by** fields. Another way to resize the root image would be to use the **Edit/Select All** menu option to select all of its shapes, and then use **Layout/Group** to make a group. Dragging the square resize handles on the green selection box will resize this group. When the root image is appropriately sized, use **Layout/Ungroup** to eliminate the grouping.

3.13.8 Editing Points on a Primitive

The vertices defining a primitive can be moved, added and deleted using **Image Editor**. Clicking on a selected primitive will enable point editing for that primitive. A primitive in point edit contains a green skeleton which connects its vertices. Representing each vertex point is a hollow green square or *point handle*. The currently selected point is shown by a blue point handle.

To move a point, select and drag the appropriate point handle. To delete a point, select its point handle and use the **Edit/Delete** menu option (or press the **Delete** key). To add a new point to the primitive, click on the green skeleton and drag the mouse. When the mouse button is released, a new point is inserted between the indicated vertices.

To leave **Point Edit mode**, click on the background or another shape.

3.13.9 Defining Stacking Order or Priority

You can specify how shapes are stacked when they overlap (*stacking order)*. To move shapes in front of or behind other shapes, use the **Bring to Front** or **Send to Back** options from the **Layout** menu.

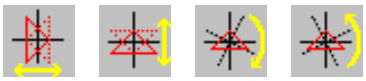
Stacking order is with respect only to other shapes in the same group or image. In other words, the **Bring to Front** menu option will bring the selected shape to the front of all other shapes in that group, but not necessarily to the front of all shapes in the window.

3.13.10 Defining the Center Point of a Shape

The center point of any image or primitive can be changed by selecting the shape, then using the **Edit/Recenter** menu option. A set of green cross-hairs will appear showing the current center point. Point to where you want the center point of the object to be, and click. To leave the **Recenter** mode, press either the **OK** or **Cancel** buttons on the dialog box.

You can reset the center point of the entire drawing (root image) by using the **Edit/Image** menu option.

3.13.11 Using the Flip and Rotate Tools



Any selected shape can be rotated about its center point by any amount. To do this, select the shape(s) and then use the **Edit/Rotate/Clockwise** or the **Edit/Rotate/CounterClockwise** menu options. If you want to set the angle by which an object is rotated, use the **Edit/Rotate/Set Angle** menu option.

To flip an object about its x-axis use the **Edit/Flip/Horizontal** menu option. To flip an object about the y-axis use the **Edit/Flip/Vertical** menu option. Remember that the intersection of the x-axis and y-axis of a shape is its *center point* (defined using the **Edit/Recenter** menu option). Before flipping or rotating a shape, first make sure that its center point is defined appropriately.

3.13.12 Align and Distribute

Multiple shapes can be aligned either vertically or horizontally to the primary selection (shown enclosed by green selection handles). They can be aligned vertically with respect to either their left edge, right edge or center. Shapes can be aligned horizontally with respect to their top edge, bottom edge, or center. To align, first select multiple objects using the **Shift** key, and then use the **Layout/Align** menu option. Select an alignment scheme from the resulting dialog box.

The **Layout/Distribute** menu option allows you to distribute three or more shapes in relation to each other. Shapes can be distributed *horizontally* so that the same space exists between left and right edges of adjacent shapes. Distributing *vertically* will reposition the shapes so that the same space exists between the bottom and top edges of adjacent shapes. Shapes can be distributed uniformly along the circumference of a circle.

3.13.13 Using Grid Lines

A grid can be used to perform precise positioning and sizing of shapes, by breaking the editor window up into divisions. You can show (or hide) grid lines by toggling the **View/Grid** menu option.

You can change the color of the grid by selecting a color from the **Color** palette and then using the **View/Grid Color** menu option. The granularity of the grid can be adjusted using the **View/Grid Spacing** menu option. By toggling the **View/Snap** menu option, you can restrain positioning and resizing of shapes to the intersections of the grid. Using the **View/Grid Spacing** option you can define the grid line interval by specifying either the total number of grid lines, or the distance (in real world coordinates) between successive grid lines.

If the **Snap** mode is active, the **View/Snap From** menu option allows you to specify which corner of a shape's bounding box will be aligned to the grid intersections during repositioning. If **View/Snap from/Center** is selected, a repositioned shape's center point will be glued to the grid intersections.

3.13.14 Changing Views (Panning and Zooming)

If working on a highly detailed portion of the image, you may want to magnify a portion of the window. To zoom in to some area of the window, first select the **View/Zoom In** menu option. Drag out a rectangle with the mouse over the area of detail. When the mouse button is released, the area inside the rectangle will be expanded to encompass the entire window. To zoom back out, use the **View/Zoom Out** menu option.

When zoomed in, you can pan to other areas of the window using the horizontal and vertical scroll bars.

Return to the default view by using the **View/View [1:1]** option. Unless the window is square, the top or bottom portion of the view may not be visible. To see the entire coordinate space, use the **View/Fit in Window** option. This viewing mode will leave dead space off to the right of the window, but guarantee the entire coordinate space will be seen.

3.13.15 Changing Dimension (Coordinate Space Boundaries)

Coordinate space boundaries (or *dimension*) can be assigned to the editor window. The default coordinate space is the common *Normalized device coordinates* or (xlo=0, ylo=0, xhi=32767, yhi=32767). These dimensions determine an object's coordinate system when it is saved. The dimension should be set to match the world coordinate system used within the program. This ensures that the positions of shapes defined from the **Image Editor** will

remain the same when they are displayed within that program. Use the **Layout/Dimension** menu option to change the dimension in the **Image Editor**.

The **Allow Icons to Scale...** check box specifies the rule defining how the image is scaled when used in the application program. If this item is checked, the image will automatically be scaled according to the world coordinate system defined by the application program. If this item is not set, the shape will stay the same size no matter what world it is attached to.

To see the current location of the pointer with respect to the editor's dimension, toggle the **View/Coordinates** menu option. The pointer's (x,y) location will be displayed in the status bar at the lower right-hand corner of the editor window.

3.13.16 Changing the Layout Size and Color

To change the editor window's background color, select the desired color from the **Color** palette and then use the **Layout/Layout Color** menu option.

If you want to increase the size of the editing area beyond what is defined by the boundaries of the world coordinate system, use the **Layout/Layout Size** menu option. A dialog will be displayed allowing you to increase the number of "screens," thereby adding space to the right and bottom sides of the editing area. This new space can be scrolled to using the right and bottom scroll bars attached to the editor window.

3.13.17 Program Access

Any image or primitive added to the root image can be accessed from inside an application by specifying a **Reference** or **Field** name through the **Properties** dialog box.

You can define whether a primitive's color can be defined programmatically through the **DCOLOR.A** attribute of its display entity field. For example, if a primitive is "definable" and the display entity pointer is ICON.PTR, your program can set the primitive's color to "15" as follows:

```
Let DCOLOR.A(ICON.PTR) = 15
```

3.14 Using the Graph Editor

The **Graph Editor** can be used to create and edit a variety of graphical objects whose purpose is to depict a single value or set of numerical values. 2-D plots, pie charts, clocks, level meters, dials, and digital displays are some of the graph objects that can be created. Graphs are not built by the user as in the **Image Editor**. Instead you start off with a template which can be modified as necessary.

3.14.1 Style, and Color Palettes

The **Style** palette on the right hand side of the window contains the set of dash styles, hatch styles, line widths, and text fonts that can be applied to the selected graph components. The

Color palette on the bottom of the window contains 64 colors that can be applied to a component. When a component is selected, the **Style** and **Color** palettes will be updated to reflect the style and color of that graph part. At this time, **Style** and **Color** palette changes will be applied to the selected part.

3.14.2 Selecting, Moving, and Resizing

Graph parts are selected by clicking the mouse button over a visible portion. Selected parts are marked by a bordering green or cyan box. Multiple components can be selected by holding down the **Shift** key and clicking on several parts. You can also select multiple components by clicking in the background of the window and dragging the mouse over the parts you want to select.

For resizing, it is necessary to select the entire graph. Use the **Edit/Select All** menu option or drag the **Select** rectangle over the whole graph. Sides and corners of the selection box contain eight small, square resize handles. Resizing is performed by clicking on down and dragging the appropriate resize handle.

To move the graph, select the graph or any of its components and drag it to the desired location.

3.14.3 Charts (2-D Plots)

A chart is a 2-D plot used to display one or more data sets represented as histograms, bar graphs, surface charts, or simple plots of 2-D data. Charts have one x-axis, one or two y axes, data sets, a title, and an optional legend.

3.14.3.1 Modifying Chart Attributes

To modify the title, legend display, or any attribute of the chart itself, select the title. Then use the **Edit/Properties** menu option. The **Chart Detail** dialog box will be displayed. It contains the following information:

- **Library Name** – The name used to load the chart into your application program.
- **Title** – The title shown on the top of the chart.
- **Axes on Edges** – If checked, numbering and tic marks will be forced to appear on the edges of the plot area. For better visual reference, two extra axes will be drawn on both the top and right sides of the plot area.
- **Time Trace Plot** − Setting this item implies that the chart is a time trace plot. Whenever a variable being monitored by the chart is modified, its new value is plotted along the Y-axis and the current *simulation time* is plotted along the X-axis.

- **Show Legend** – Chart will show a legend below the plot area. The fill style and color of each data set is shown preceding its name.
- **Show Border** – A chart can be defined to draw a rectangular background underneath.
- **Data Sets** – A data set can be added using the **Add** button, or removed by selecting its name in the list box and then pressing the **Remove** button. To change the name of a data set, select its current name in the list box and then press the **Edit** button. (see "Attributes of a Data Set")
- **Handling of Multiple Data Sets** – If "stacked," all discrete data sets will be stacked on top of each other. In other words, the value plotted in a data cell is reflected as the *height* of the bar, not its top. Therefore, *stacking* means that the bottom of a cell in data set *n* is equal to the top of the same cell in data set *n-1*. I.e. higher numbered data sets are stacked onto the lower numbered ones.

3.14.3.2 Modifying the X-Axis

To change the range, numbering interval, or any other property associated with the X-axis, first choose the axis (or one of its components), double click on the axis or choose the axis (or one of its components), and use the **Edit/Properties** menu option. The X-axis has the following properties:

- **Title** – Label for X-axis displayed below numbering.
- **Rescaleable** – Specifies whether the X-axis will be re-numbered (scaled) when one of the data points extends beyond its limit. In this case, the **Compress Data** item determines whether a scrolling window is used, and whether old data is discarded, or the range of the graph is to be expanded showing all data. Note that rescaling may modify the tic mark, numbering, and grid line intervals to maintain a similar visual representation of the chart. If this item is not checked, data points falling beyond the limits of the X-axis will be discarded.
- **Show Grid Lines** – If the this item is on, *grid lines* will be shown crossing the X- axis.
- **Tics Centered, Tics Inside, Tics Outside** – Defines the tic mark alignment with respect to the X-axis line. Tics marks can be attached to the X-axis from their center, left or right sides.
- **Compress Data** – When this item is set, re-scaling the X-axis will increase the coordinate area of the chart enough to encompass the offending data point. As a result, existing data will shrink in size. Clearing this item will have data *scrolled* along the X-axis during axis rescale. In this case, data scrolled out of view will be discarded.
- **Minimum, Maximum** – Defines the initial X-axis data range of the chart.
- **Tic Interval (Major & Minor)** – Defines the distance along the X-axis between consecutive tic marks. If an interval is zero, tic marks will not be displayed.
- **Numbering Interval** – Defines the distance along the X-axis between consecutive number labels on the axis.

- **Grid line Interval** – Defines the distance along the X-axis between consecutive grid lines.
- **Y Intersection Point** – Defines the point (in x-axis coordinates) along the X-axis where the Y-axis intercepts.
- **Y2 Intersection Point** – Defines the point (in x-axis coordinates) along the X-axis where the second Y-axis intercepts.
- **Data Scaling Factor** – Defines the factor multiplied to the X component of all data plotted to the chart at runtime.

3.14.3.3 Modifying the Y-Axis

To change the range, numbering interval, or any other property associated with the Y-axis, double click on the axis or choose the axis (or one of its components), and use the **Edit/ Properties** menu option. The Y-axis has the following properties:

- **Title** – Label for Y-axis displayed to the left of its numbering.
- **Rescaleable** – Specifies whether the Y-axis will be re-numbered (scaled) when one of the data points extends beyond its range. Note that re-scaling may modify the tic mark, numbering, and grid line intervals to maintain a similar visual representation of the chart. If this item is not checked, data points falling beyond the limits of the Y-axis will be clipped.
- **Show Grid Lines** – If this item is on, *grid lines* will be shown crossing the Y-axis.
- **Tics Centered, Tics Inside, Tics Outside** – Defines the tic mark alignment with respect to the Y- axis line. Tic marks can be attached to the Y- axis from their center, left or right sides.
- **Minimum, Maximum** – Defines the initial Y-axis data range of the chart.
- **Tic Interval (Major & Minor)** – Defines the distance along the Y-axis between consecutive tic marks. If an interval is zero, tic marks will not be displayed.
- **Numbering Interval** – Defines the distance along the Y-axis between consecutive number labels on the axis.
- **Grid Line Interval** – Defines the distance along the Y-axis between consecutive grid lines.
- **X Intersection Point** – Defines the point (in y-axis coordinates) along the Y-axis where the X-axis intercepts.
- **Data Scaling Factor** – Defines the factor multiplied to the Y component of all data plotted to the chart at runtime.

3.14.3.4 Modifying the Second Y-Axis

To change the range, numbering interval, or any other property associated with the second Y-axis, double click on the axis or choose the axis (or one of its components), and use the **Edit/Properties** menu option. The second Y-axis has the following properties:

- **Title** – Label for Y-axis displayed to the left of its numbering.

- **Rescaleable** – Specifies whether the Y-axis will be re-numbered (scaled) when one of the data points extends beyond its range. Note that re-scaling may modify the tic mark, numbering, and grid line intervals to maintain a similar visual representation of the chart. If this item is not checked, data points falling beyond the limits of the second Y-axis will be highlighted.
- **Show Grid Lines** – If the this item is on, *grid lines* will be shown crossing the second Y-axis.
- **Tics Centered, Tics inside, Tics Outside** – Defines the tic mark alignment with respect to the second Y-axis line. Tics marks can be attached to the Y-axis from their center, left or right sides.
- **Minimum, Maximum** – Defines the initial data range of the second Y-axis.
- **Tic Interval (Major & Minor)** — Defines the distance along the second Y-axis between consecutive tic marks. If an interval is zero, tic marks will not be displayed.
- **Numbering Interval** – Defines the distance along the second Y-axis between consecutive number labels on the axis.
- **Grid Line Interval** – Defines the distance along the second Y-axis between consecutive grid lines.
- **Data Scaling Factor** – Defines the factor multiplied to the Y component of all data plotted to the chart at runtime.

3.14.3.5 Attributes of a Data Set

You can edit individual attributes of a data set by selecting the bars or plot line of the desired data set and using the **Edit/Properties** menu option. Its **Detail Dialog** detail includes:

- **Representation** – Defines how the overall data set is structured. You can choose one of the following data set types:

    1. **Bar Graph** – Contains a fixed number of cells. Each new data point changes the nearest cell's plot. Neighboring cells are NOT connected. The first cell begins at (X_Minimum - Cell_Width / 2) units. The individual bar is centered over the cell, and there is a small gap between bars.

    2. **Histogram** – Also contains a fixed number of cells. Each new data point changes the nearest cell's bar. There is no connection between neighboring cells. The bar is set at the left edge of the cell, and there is no gap between bars. The first data cell begins at the X-axis minimum.

    3. **Discrete Surface –** Neighboring cells are connected to form a surface, however there are still a fixed number of cells. Each new data point changes the nearest "peak or valley" on the surface. The first cell begins at (X_Minimum - Cell_Width / 2) units.

    4. **Continuous Surface** – Variable number of cells, i.e. a new cell is added to the graph each time a data point is plotted at the given (x,y) location. Neighboring cells are connected.

- **Plot Type** – A data set can be shown using a filled region or a simple surface line with or without markers:

  1. **Fill** – Plot a data cell using a filled polygon. The fill style can be reset using the **Style** palette .
  2. **Line** – Plot data cell using a polyline. Use the **Style** palette to reset the line width or dash style.
  3. **Marker** – Use a small marker to represent the data point. The specific marker used for the data point is determined from the **Edit/Mark Style** menu option menu. Markers are only valid for the "continuous surface" representation.

- **Cell Width** – For bar, histogram and discrete surface data sets, this is the size of each data cell. For histograms, the first data cell begins at the X-axis minimum. For bar and surface graphs, the first cell begins at (X_Minimum - Cell_Width / 2) units.

- **Interpolate** – This check box determines whether there is linear interpolation in forming the connecting surface between consecutive data points. If this item is NOT checked, the surface will be shown with only horizontal and vertical lines.

- **Use Left Axis / Use Right Axis** – Your chart can be defined to simultaneously show two sets of independently scaled data by using a second Y-axis (generally shown to the right of the plot area). Each data set in your chart can belong to either the left or right (second) Y-axis.

- **Static** – This item is used to enhance performance whenever you do not intend the plot to be modified once it has been displayed. In this case, a single polygon (or polyline) will be used to display all cells in the data set.

3.14.3.6 Creating a Time Trace Plot

If you want the chart to be used to plot the value of a single variable over simulation time, a time trace plot should be used. To create a trace plot, select the graph and use the **Edit/ Properties** option. Set the **Time Trace Plot** checkbox in the **Chart Detail Dialog**.

3.14.4 Pie Charts

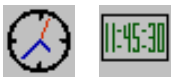
A pie chart can depict a fixed sized array of scalar values in relation to one another. By selecting the **Pie Chart** and using the **Edit/Properties** menu option you can change the names and initial values shown by each pie slice. The color and fill style of individual slices and other components (including legend text, title, and borders) can be changed by selecting them and using the **Style** or **Color** palettes. The **Detail Dialog** for a pie chart contains the following:

- **Library Name** – The name of the object within the current graphics library.

- **Title** – Text of title displayed on top.
- **Show borders** – Determines whether to put borders around the legend, title, and plot of a pie chart.
- **Slice List Box** – This list box contains the names of all slices in the chart.

  1. To add a slice, set the new slice's name and value in the **Slice Name** and **Slice Value** text boxes below, and press the **Add** button.
  2. To remove a slice, select its name in the list box and press the **Remove** button.
  3. To change the name or value of a slice, first select its name in the list box, and then update the **Slice Name** and **Slice Value** text boxes and press the **Update** button.

3.14.5 Clocks



Clocks are used to display simulation time within a program. Both analog and digital varieties of clocks are available. By selecting the clock and using the **Edit/Properties** menu option you can change its various attributes including axis scaling parameters as well as whether or not to display hours, minutes and seconds. The color and fill style of individual components (including face, title, and border) can be changed by selecting them and using the **Style** or **Color** palettes. The **Detail** dialog for a clock contains the following:

- **Library Name** – The name of the object within the current graphics library.
- **Title** – Text of title displayed on bottom.
- **Interval** – (Analog clock only) Distance between tic marks around the face.
- **Num Interval** – (Analog clock only) Distance between numbers around the face**.**
- **Max Hours** – The maximum number of hours the clock (shown at the top of the face) that the clock is capable of showing (generally 12). As this value is exceeded, the time display will start over from 0:00:00.
- **Show Hours, Show Minutes, Show Seconds** – You can control displaying the hour, minute and second hands with these items.
- **Hours Per Day** – Currently, this parameter has no effect on the layout of the clock. It is only used within the application program.
- **Minutes Per Hour** – Defines the time interval before the "hours" are incremented by one.
- **Seconds Per Minute** – Defines the time interval before "minutes" are incremented.
- **Show Borders** – (Analog clock only) Determines whether to put borders around the legend, title, and plot of a pie chart.

3.14.6 Dials

A dial can be created in the **Graph Editor** for displaying a single scalar value. The hand of the dial rotates clockwise as its value gets larger. By selecting the dial and using the **Edit/ Properties** menu option you can change the various attributes shown below:

- **Library Name** – The name of the object within the current graphics library.
- **Title** – Text of title displayed on bottom.
- **Minimum, Maximum** – Defines the range of values shown by the dial.
- **Interval** – Distance between tic marks around the face.
- **Num Interval** – Distance between numbers around the face**.**
- **Min Theta** – Angle in degrees where the minimum value is placed around the dial circumference.
- **Max Theta** – Angle in degrees where the maximum value is placed around the dial circumference.
- **Scale Factor** – Factor multiplied by value before being displayed in the dial.
- **Show Border** – A square background can be shown under the dial face and title.

3.14.7 Level Meters

A level meter shows a single scalar numerical value. The level meter is composed of a bar which grows and shrinks along a vertical axis. The height of the bar reflects the magnitude of the value being plotted. By selecting the meter and using the **Edit/Properties** menu option you can change the attributes shown below:

- **Library Name** – The name of the object within the current graphics library.
- **Title** – Text of title displayed on bottom.
- **Minimum, Maximum** – Defines the range of values shown by the meter.
- **Interval** – Distance between tic marks along the axis.
- **Num Interval** – Distance between numbers along the axis**.**
- **Show Grid LIines** – Horizontal grid lines extending across the plot area can be shown.
- **Scale Factor** – Factor multiplied by value before being displayed in the meter.

3.14.8 Digital Displays

A digital display is for showing a single scalar numerical value. The value is shown explicitly as numerical text and is enclosed by a box. By selecting the display and using the **Edit/ Properties** menu option you can change its various attributes shown below:

- **Library Name** – The name of the object within the current graphics library.
- **Title** – Text of title displayed on bottom.
- **Minimum, Maximum** – Defines the range of values shown by the meter.
- **Field Width** – Number of places allotted for the entire value (including decimal point).
- **Precision** – Number of places to the right of the decimal point. If zero, an integer value is shown.
- **Scale Factor** – Factor multiplied by value before being displayed in the meter.

3.14.9 Text Meters

This is a titled text value enclosed by a box. The following attributes can be set:

- **Library Name** – The name of the object within the current graphics library.
- **Title** – Text of title displayed on bottom.
- **Width** – Number of places allotted for the text value.

3.15 Using the Dialog Editor

The **Dialog Editor** (figure 3-3) provides a fast and easy to use drag and drop facility for creating and editing dialog boxes. A dialog box is a container for controls which accept various types of input. A dialog box can contain buttons, single and multi-line text boxes, combo boxes, value boxes, list boxes, radio boxes, check boxes, text labels, and tables. Tabbed dialog boxes can also be created. Items contained by a dialog box or a dialog box tab are called *controls.*

Controls are created and added to the dialog box via the **Mode** palette on the left-hand side of the window. To create a control, first select the control type from the **Mode** palette. Position the pointer over where you want the control to go into the dialog box and press the

mouse button. The dialog box will automatically resize as needed to fit the controls it contains. It is OK to drop a control *outside* of the dialog box in order to make the box bigger.

The actual dialog box you are working on can be displayed using the **Layout/Show Dialog** menu. Double click on the "**-**" in the header bar of the dialog window to make it disappear.

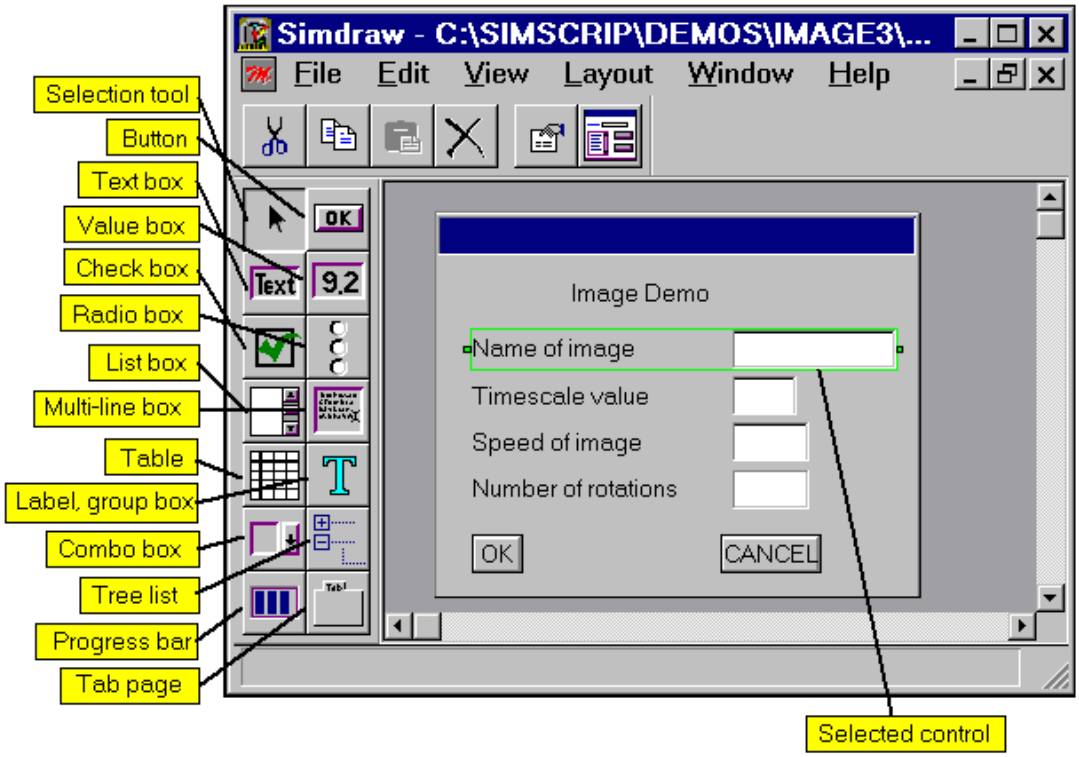


Figure 3-3. Dialog Editor

3.15.1 Selecting, Moving, and Resizing

Selected controls are marked by a bordering green or cyan box. Sides and corners of this box may contain small square resize handles. A resize handle is present for each dimension that the control can logically be resized in. Resizing is performed by clicking down and dragging a resize handle.

To move a control, click down on it and drag to the desired location.

3.15.2 Dialog Box Coordinate System

Controls are positioned in *font units*. The width of a font unit is the width occupied by a single digit within a dialog box. The height of a font unit is the maximum of button and text box heights. The origin of a dialog box's coordinate system is at its top left-hand corner with Y-values increasing downward.

3.15.3 Using the Clipboard (Cut, Copy, Paste Commands)

The **Dialog Editor** supports the standard **Cut, Copy,** and **Paste** operations found under the **Edit** menu option. The **Cut** option deletes selected controls and places them in the clipboard. The deleted item remains on the clipboard until the next time you use the **Edit/Cut** or **Edit/Copy** option. Use the **Edit/Paste** option to paste as many copies as you want from the clipboard into the image. Controls can be deleted without changing the clipboard by using the **Edit/Delete** option.

The clipboard is shared among all active **Dialog Editor** sessions. You can copy graphics from one image into another by activating the source edit window, using the **Copy** option, and activating the destination editor and using the **Paste** option.

Note: The dialog box itself can never be "cut" or "deleted". It can, however, be selected for the purpose of changing its properties.

3.15.4 Controls

To create a control (check box, button, text box, etc.) select the appropriate control from the **Mode** pallette and drag its outline to where you want it to go on the dialog box. All controls have the following attributes:

- **X, Y Position** – Position in font units from the upper left-hand corner of the dialog box.
- **Reference (Field) name** – Any control added to the dialog can be accessed from inside an application by specifying a **Reference** or **Field** name.

Buttons

A button receives simple input and contains no data from the user. Using the **Edit/Properties** menu option you can set the following attributes of a button:

- **Label** – This is the text shown on the face of the button.
- **Default** – Setting this item will make this button the "default" button. This button will be pressed when you press the **Enter** key.
- **Verifying** – This will cause the button to check the contents of all value boxes in the same dialog when it is pressed.
- **Terminating** – Setting this check box will make the button erase its dialog box when pressed.

Text Boxes

Text boxes are used to receive single line text string input. Using the **Edit/Properties** menu option you can set the following attributes of a text box:

- **Label** – The text appearing on the left-hand side of the box.
- **Width** – The number of characters that the text box can show.
- **Text** – The text string initially shown in the box.
- **Selectable Using Return** – Defines whether the application program will be notified when you press the **Return** key while this text box has input focus.

Value Boxes

A value box is used to receive or show a single numerical value to the user. Using the **Edit/Properties** menu option you can set the following attributes of a value box:

- **Label** – The text on the left-hand side of the box identifying value type to the user.
- **Min** – The minimum value the box can contain. If a value typed into the box is out of range, the user will be informed whenever a *verifying* button is pressed.
- **Max** – The maximum value the box can contain.
- **Precision** – Precision is used to format output and round input. It defines the number of digits to the right of the decimal point. (0 = integer value, 1 = 0.1, 2 = 0.01, -1 = rounded to 10's, -2 = rounded to 100's etc.)
- **Value** – The initial value displayed in the value box.
- **Use Scientific Notation** – Indicates whether output should be formatted using scientific notation. (i.e. 71 = 7.1e+1).
- **Selectable Using Return** – Defines whether the application program will be notified when the user presses the **Return** key while this text box has input focus.

Check Boxes

A check box is used to receive and show yes/no input. Using the **Edit/Properties** menu option you can set the following attributes:

- **Label** – The text on the right-hand side of the box identifying it to the user.
- **Checked** – Initial state of the check box.

Radio Boxes

The radio box accepts input from a fixed list of alternatives. It contains a set of *radio buttons*. You can only select one radio button at a time; when you select a new button, the previously selected button pops up automatically. You can add and remove radio buttons from the radio box using the **Edit/Properties** menu option:

- To add a button, enter its *label*, and *reference name* in the **Radio Buttons** area of the **Properties** dialog, and then press the **Add** button.
- To remove a button, select its label in the list box and then press the **Remove** button.
- To change the attributes of a button, select its label in the list box, modify its label, or reference name, and then press the **Update** button.

List Boxes

A list box is used to accept input from a list of text values. The list may vary in length and will be scrollable, if needed. You can define the list to accept only single item selections, or accept multiple item selections using the **Shift** and/or **Ctrl** keys. Using the **Edit/Properties** menu option you can set the following attributes:

- **Width** – The width in font units of the list (including scroll bars).
- **Height** – The height in font units of the list.
- **Allow Multiple Selections** – Allows the user to select several items in the list using the **Shift** and **Ctrl** keys.

Multi-line Text Box

A multi-line text box can receive and show unlimited lines of text. Horizontal and vertical scroll bars are attached, if needed. You can easily edit the text it contains using the mouse. Using the **Edit/Properties** menu option you can set the following attributes:

- **Width** – The width in font units of the box (including scroll bar).
- **Height** – The height in font units of the box (including scroll bar) .
- **Text** – The text initially displayed in the box.
- **Allow Horizontal Scrolling** – If checked, a horizontal scroll bar will be attached whenever a line of text is too long to be viewed in the text box. If not checked, long text lines will be truncated.

Labels & Group Boxes

A label is used to place explanatory text or titles in a dialog box. It can be positioned anywhere within the dialog. A group box can be attached to the label and sized to enclose a set of controls with some common property. Using the **Edit/Properties** menu option you can set the following attributes:

- **Label** – The text of the label.
- **Show Group Box** – Defines whether a group box will be shown.
- **Width** – The width in font units of the group box.
- **Height** – The height in font units of the group box.

Through the **Properties** dialog, you can define whether the label is defined programmatically through the **DTVAL.A** or **DDVAL.A** attributes of its field pointer. One of the following three access modes can be defined:

a. Use the **DTVAL.A** attribute to define the text.
b. Use the **DTVAL.A** attribute to define the text. Truncate the text to **Field width** places.
c. Use the **DDVAL.A** attribute to define a real value displayed by the label. The **Field width** text box specifies the total number of places, while the **Precision** box defines the number of places after the decimal point.

For example, if the label's reference name is "MY.LABEL", you could programmatically set the label as follows:

```
Let DTVAL.A(DFIELD.F("MY.LABEL", FORM.PTR)) = "Hello World"
```

or

```
Let DDVAL.A(DFIELD.F("MY.LABEL", FORM.PTR)) = 12.5
```

Combo Boxes

A combo (combination) box is a text box containing a small "drop down" button. When that button is pressed, a scrollable list of choices for the text field is displayed. The combo box can be defined to allow only those alternatives shown in the list entered, or to be fully editable like a text box. Using the **Edit/Properties** menu option you can set the following attributes:

- **Label** – The text on the left-hand side of the box identifying the box.
- **Width** – The width in font units of the text box plus the drop down button.

- **Height** – The number of visible items in the drop down list.
- **Editable** – Defines whether or not you can edit the text field, or, if it is restricted, to contain only one of the items shown in the drop down list.
- **Sorted Alphabetically** – If checked, items in the drop down list will be shown in alphabetical order.

Progress Bar

A progress bar is a programmatically adjustable horizontal bar usually used to indicate the completion status of a task. The length of the bar is proportional to the value given to it by the program. The bar cannot be adjusted by the user, only by the program. Using **Edit/ Properties** menu option you can set the following attributes of the bar:

- **Label** – The text on the left hand side of the bar identifying it to the user.
- **Width** – The maximum visible size of the bar in font units.
- **Min** – The bar will have zero length when set to this value.
- **Max** – The bar will have maximum length when set to this value.
- **Value** – The initial value displayed by the bar.

Tables

A table is a two dimensional array of selectable text fields or "cells". The table can be horizontally and vertically scrollable. All cells in the same column have the same width, but you can define the width of this column.

A table can have both column and row headers. A row of "column headers" is shown on top of the array of cells. This special row of cells will scroll horizontally with the rest of the table, but not vertically. "Row headers" are shown in a column to the left of the table. This column scrolls with the table only in the vertical direction.

You can navigate through a table using the left-, right-, up- and down-arrow keys. The program will be informed of cell selection whenever an arrow key is used to move to a different cell. You can tell the table to automatically add a new row of cells to its bottom row whenever the you attempt to move past the last row using the down-arrow key. Use the **Edit/Properties** menu option to set the following attributes:

- **Viewed Width** – The total width in font units of space occupied by the entire table (including row headers, and scroll bar).
- **Viewed Height** – The total height in font units of space occupied by the entire table (including column headers and scroll bar).

- **Number Columns** – Number of columns of cells (not including headers).
- **Number Rows** – Number of rows of cells (not including headers).
- **Column Headers** – If checked, the table will contain a separate row of column headers at the top of the cells.
- **Row Headers** – If checked, the table will contain a separate column of row headers on the left of the cells.
- **Automatic Grow** – If checked, the table will automatically add a row, if the you attempt to move past the last row with the "down-arrow" key.

The attributes of all columns in the table are shown within a separate **Column Detail** table invoked by clicking on the **Columns** button:

- **Column (1,2,...) Width** – The number of characters shown in the cells of a particular column. Select the cell in the column corresponding to the one you want to change, and type in a new width.
- **Column (1,2,...) Alignment** – Text in a table cell can be justified to the left or right, or can be centered. Within the **Column Detail** table (**l**=Left justified, **c**=Centered, and **r**=Right justified).

You can also set the initial contents of the cells in the table by clicking on the **Contents ...** button. A duplicate table of the one you are working on will show the initial contents of all cells. To change the initial contents of a cell, select the corresponding cell in the **Cell Detail** table, and then type in the new text and press **Return**.

Dialog Box

Although the dialog box annotation cannot be moved or resized, it can still be edited by selecting it and using the **Edit/Properties** menu option. The dialog box can be defined with the following attributes:

- **Library Name** – The name used to access the dialog box from inside the application.
- **Title** – The text shown on the header bar of the dialog.
- **Modal Interaction** – Defines whether the dialog is "modal" or "modeless". When a modal dialog box is displayed, the user cannot interact with any other component of the application but the contents of that dialog box. Modeless dialogs can be interacted with asynchronously.
- **Position with Respect to Screen** – Specifies which corner of the dialog will be offset from the lower left-hand corner of the screen. For example, if **Bottom Right** positioning is selected, the **X Offset** and **Y Offset** fields define the distance from the bottom left-hand corner of the screen to the bottom right corner of the dialog box. This distance is specified in "screen coordinates" where the width and height of the computer screen each are each 100 units.

- **Tab Ordering of Members** – If you wish to use the **Tab** key to transfer input focus from one control to the next while interacting with the dialog box, the order in which this traversal takes place can be established ahead of time. A list box shows the labels of all controls in the dialog that can have input focus. The order of items in this list is the order in which input focus will proceed when the **Tab** key is pressed. Use the up- and down-arrow keys to shift the tab ordering of controls.

You can also define how **ACCEPT.F** will behave when displaying the form. See paragraph 5.3.2.

Tabbed Dialogs

The **Dialog** editor can be used to create **Tabbed Dialogs** or to convert existing dialogs to be tabbed. Using a **Tabbed Dialog** you can attach sets of controls to overlapping **Tab Fields**. Only the top **Tab Field** can be seen; all other tab fields and attached controls remain hidden underneath. The only visible portion of a **Tab Field** is a small rectangular area containing its name, or a *tab*. Clicking on the tab will bring the **Tab Field** to the top of the tab area and show all controls attached to it.

To create a **Tabbed Dialog**, you must first make sure that the area on the dialog box where the **Tab Field** is to be placed is cleared of controls (they should be moved or temporarily cut to the clipboard.) Create a **Tab Field** by dragging it from the palette onto the dialog box. Any number of **Tab Field**s can be dropped onto the dialog box. The tab area can be resized by resizing the top **Tab Field**, but cannot be moved.

Dropping a control onto the top **Tab Field** will automatically attach it to that tab. Controls can be dragged from the **Mode** palette, pasted from the clipboard, or moved onto the top **Tab Field**.

The tab area is not automatically resized when controls are dropped onto the **Tab Field**. It should be sized manually prior to adding controls.

To remove a **Tab Field**, first remove all controls it contains and then use the **Edit/Cut** or **Edit/Delete** menu options. Using the **Edit/Properties** menu option you can set the following attributes of the selected **Tab Field**:

- **Label** – The text label shown on the "tab" part of the **Tab Field**.
- **Icon Name** – The resource or file name (without extension) of the bitmap shown on the front of the tab.

3.15.4.1 Converting Conventional Dialog Boxes to be Tabbed

Perform the following steps to add tabs to an existing (untabbed) dialog box.

1. Create space for the tab area by selecting all controls using the **Edit/Select All** menu option and moving them into a saved area on the dialog box (move them down or to the right by a liberal amount.)
2. Drag a **Tab** onto the dialog box from the **Mode** palette. Resize the **Tab** according to how much space it needs.
3. Move each control which must go onto this **Tab Field** from the saved area.
4. Repeat steps two and three until all **Tab**s have been created and filled with controls.
5. Select each **Tab** and use the **Edit/Properties** menu option to set the label on the **Tab**, its icon, etc.

3.15.4.2 Align and Distribute

Multiple controls can be aligned either vertically or horizontally to the primary selection (shown enclosed by green selection handles). They can be aligned vertically with respect to either their left edge, right edge or center. Controls can be aligned horizontally with respect to their top edge, bottom edge, or center. To align, first select multiple objects using the **Shift** key, and then use the **Layout/Align** menu option. Select an alignment scheme from the resulting dialog box.

The **Layout/Distribute** menu option allows you to distribute three or more controls in relation to each other. Controls can be distributed *horizontally* so that the same space exists between left and right edges of adjacent controls. Distributing *vertically* will reposition the controls so that the same space exists between the bottom and top edges of adjacent controls.

3.15.4.3 Using Grid Lines

A grid can be used to perform precise positioning and sizing of controls by breaking the editor window up into divisions. You can show (or hide) grid lines by toggling the **View/Grid** menu option.

You can change the color of the grid by selecting a color from the **Color** palette and then using the **View/Grid Color** menu option. The granularity of the grid can be adjusted using the **View/Grid Spacing** menu option. Granularity can be **Fine, Medium**, or **Coarse**:

- **Fine** – 1 font unit wide, 0.25 font units high.
- **Medium** – 2 Font units wide, 0.5 font units high.
- **Coarse** – 3 Font units wide, 1 font unit high.

By toggling the **View/Snap** menu option, you can restrain positioning and resizing of shapes to the intersections of the grid.

3.15.4.4 Changing Views (Panning and Zooming)

You may want to magnify a portion of the dialog. To zoom in to some area of the window, first use the **View/Zoom In** menu option. Then drag out a rectangle with the mouse over the area of detail. When the mouse button is released, the area inside the rectangle will be expanded to encompass the entire window. To zoom back out, use the **View/Zoom Out** menu option.

When zoomed in, you can pan to other areas of the window using the horizontal and vertical scroll bars.

You can return to the default view by using the **View/View [1:1]** menu option. Unless the window is square, the top or bottom portion of the view may not be visible. To see the entire coordinate space, use the **View/Fit In Window** menu option. This viewing mode will leave dead space off to the right of the window, but guarantees the entire coordinate space will be seen.

3.15.4.5 Changing the Layout Size, Color and Font

To change the editor window's background color, use the **Layout/Set Color...** menu option. Select the RGB values of the background color.

Use the **Layout/Set size...** menu option if you want to increase the size of the editing area to allow you to create or edit very large dialog boxes. A dialog will be displayed allowing you to increase the number of "screens," thereby adding space to the right and bottom sides of the editing area. This new space can be "scrolled" to using the right and bottom scroll bars attached to the editor window.

The font used to depict labels and other text shown in a dialog can be reset with the **Layout/Set Font...** menu option. To have the icons representing your controls appear smaller or larger, simply select a smaller or bigger font.

3.16 Using the Menu Bar Editor

A menu bar contains *menus* which can contain either menu items, or other menus. The **Menu Bar Editor** (figure 3-4) allows you to construct a menu bar by interactively dragging and dropping icons representing menus and menu items onto a menu bar icon.

Menus and menu items items are created and added to the menu bar via the **Mode** palette on the left-hand side of the window. To create a menu, first press the **Menu** button on the **Mode** palette. Position the pointer over where you want the menu to go onto the menu bar and press the mouse button. The menu will automatically be inserted into the menu bar.

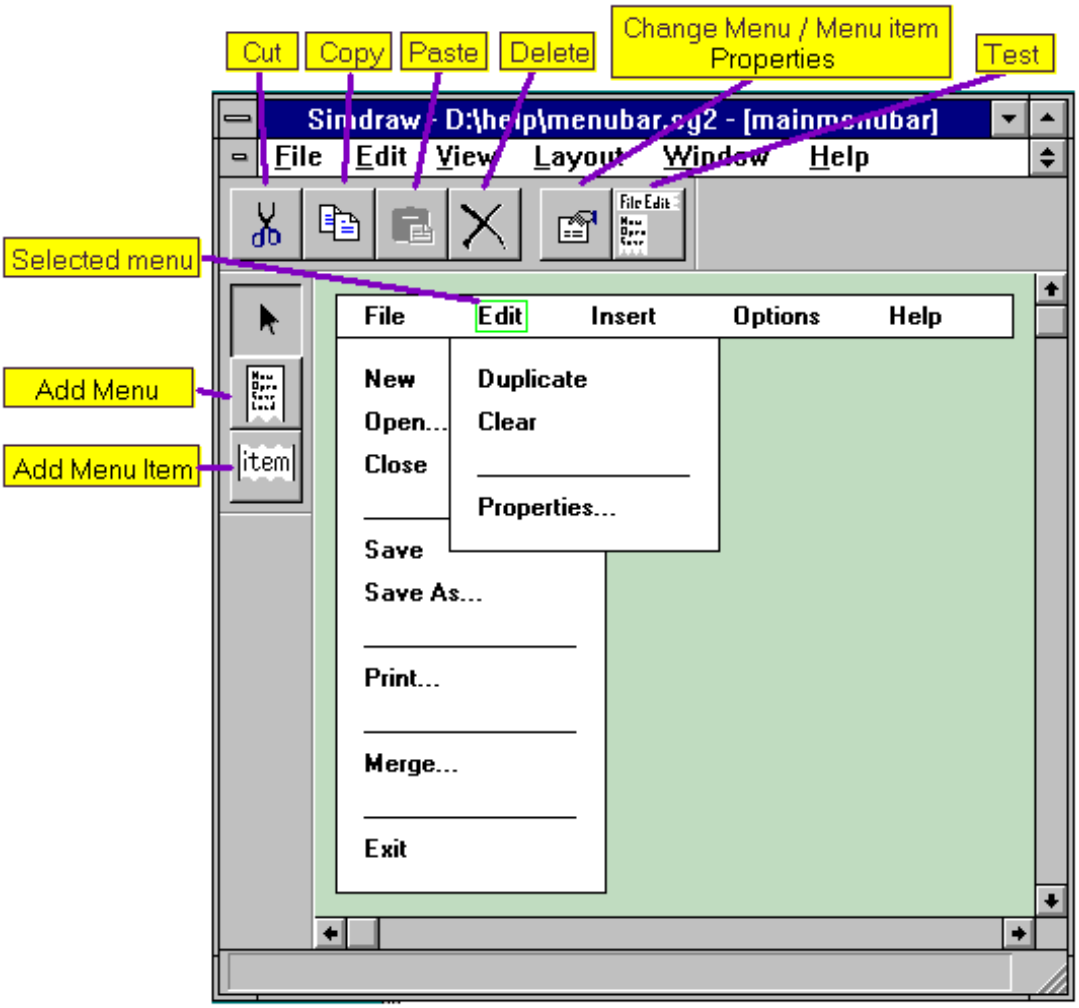


Figure 3-4. Menu Bar Editor

Menu panes can be displayed by simply clicking on the menu label. Unlike a “real” menu bar, multiple menu panes can be dropped down at the same time allowing you to transfer their menu items from one menu to another. A new menu item can be added to a menu by first dropping down the menu pane, and then dragging a menu item from the **Mode** palette to the position in the pane where you want it to go.

A usable menu bar can be interacted with using the **Layout/Show Menu Bar** menu option. A temporary window will be displayed containing a “test” menu bar. Double click on the "**-**" in the header bar of the temporary window to make it disappear.

3.16.1 Selecting and Moving (Transferring)

A menu or menu item can be selected by clicking the mouse button over its label. Selected menus are marked by a bordering green or cyan box. Selecting the label of a menu will drop down its pane, showing all the items it contains. Multiple items can be selected by holding down the **Shift** key and then clicking on several items. To add a menu or item to another menu, drop it onto the menu's open pane.

You can also select the menu bar and edit its properties, but the bar cannot be moved. You are not allowed to resize menus or the menu bar; they are resized automatically when new items are added to them.

3.16.2 Using the Clipboard (Cut, Copy and Paste Commands)

The **Menu Bar Editor** supports the standard cut, copy, and paste operations found under the **Edit** menu. The **Cut** option deletes selected items and places them in the clipboard. The deleted item remains on the clipboard until the next time you use the **Edit/Cut** or **Edit/Copy** options. You can use the **Edit/Paste** option to paste as many copies as you want from the clipboard onto any open menu pane. Items can be deleted without changing the clipboard by using the **Edit/Delete** option.

The clipboard is shared among all active **Menu Bar Editor** sessions. You can copy graphics from one menu bar into another by activating the source edit window, using the **Copy** option, and then activating the destination editor and using the **Paste** option.

Note: The menu bar itself can never be cut, copied, or deleted. It can, however, be selected for the purpose of changing its properties.

3.16.3 Editing the Menu Bar

The menu bar is not movable or resizeable, but using the **Edit/Properties** menu option you can modify the **Library Name** of the menu bar.

You can also define how **ACCEPT.F** will behave when displaying the form. See paragraph 5.3.2.

3.16.4 Editing a Menu

You can add menus to the menu bar or other menus by dragging and dropping. The menu's pane can be displayed or hidden by clicking on its text label within its container. A menu is defined by the following parameters:

- **Reference (Field) name** – Any menu added to the menu bar or another menu can be accessed from inside an application by specifying a **Reference** or **Field** name. The field name is passed to the callback routine whenever a menu item is clicked on.
- **Label** – The name identifying the menu which appears within the container menu bar or menu.
- **Mnemonic** – A letter in the menu's label that can be typed from the keyboard (while holding down the **Alt** key) to bring down the menu pane. The mnemonic character will appear underscored in your application.

3.16.5 Editing a Menu Item

A menu item can only be contained on a menu pane, and cannot contain other items. Your application program is only informed of selections of a menu option, not of a menu or menu bar. Double click or use the **Edit/Properties** menu option to change the attributes of a menu option:

- **Reference (Field) name** – Any menu item can be accessed from inside an application by specifying its **Reference** or **Field** name. The field name is passed to the callback routine whenever a menu item is clicked on.
- **Label** – The name identifying the menu item appearing within the container menu.
- **Mnemonic** – A letter in the item's label that can be typed from the keyboard (while holding down the **Alt** key) to activate the item. The mnemonic character will appear underscored in your application.
- **Accelerator Key Name** – While running the application, you can use the keyboard to activate menu options instead of using the mouse. Any menu item can have its own accelerator key. This attribute determines which key will be mapped to this menu item. To use enable keys such as [a-z], [0-9], and other punctuation and symbols keys to activate the menu item, just type the key character directly. The naming convention for keys performing functions are defined below:
- **"escape"** – Names the **Esc** or **Escape** key.
- **"delete"** – Names the **Del** or **Delete** key.
- **"return"** – Names the **Enter** or **Return** key.
- **"backspace"** – Names the ← or the **Backspace** key.

- **"tab** "– Names the **Tab** key.
- **"f1", "f2", ..., "fn "–** Names the function keys **"F1", "F2", ..., "Fn"** at the top of the keyboard.
- **Use Alt, Use Ctrl, Use Shift** – Specifies which modifier key must be held down in conjunction with the accelerator key described above.
- **Accelerator Key Label** – This is the name appended to the menu item label used to describe how to invoke the keyboard accelerator. For example, the string **"(Ctrl+C)"** could describe an accelerator activated by holding down the **Ctrl** key and pressing "**C**".
- **Status Message** – If the window containing this menu bar has a status bar, this help message will appear in the first status bar pane. The text will be displayed whenever this menu item is *highlighted* by the pointer (not necessarily activated).
- **Checked** – Menu items can have an "off/on" state shown by a small check mark next to the label. The initial state is defined by the **Checked** attribute.

  Note: This state is NOT changed automatically when the item is clicked on, but must be updated by the application program.

3.17 Using the Palette Editor

The **Palette Editor** (figure 3-5) provides a fast and easy to use drag and drop facility for creating and editing palettes, toolbars etc. A palette is usually attached to the side of your window (but is sometimes a separate window) and contains an array of buttons. The face of each button can contain a bitmap icon or show a color. Separator objects can be added to the palettes to produce space between groups of buttons.

You can define the number of columns or rows that the palette contains. For palettes attached to the left and right sides of the window, or for floating palettes, the number of columns is specified. The number of rows is used for palettes glued to the top or bottom window edges.

Palette buttons and separators are created and added to the palette via the **Mode** palette on the left-hand side of the edit window. To create a palette button, first select the **Button** icon from the **Mode** palette. Position the pointer over where in the palette you want the buttons to go, and click the mouse. The palette will automatically resize as needed to fit the buttons it contains. It is OK to drop a button *outside* of the palette in order to make it larger.

The actual palette you are working on can be displayed and tested using the **Layout/Show Palette** menu option. Double click on the "**-**" in the header bar of the palette test window to make it go away.

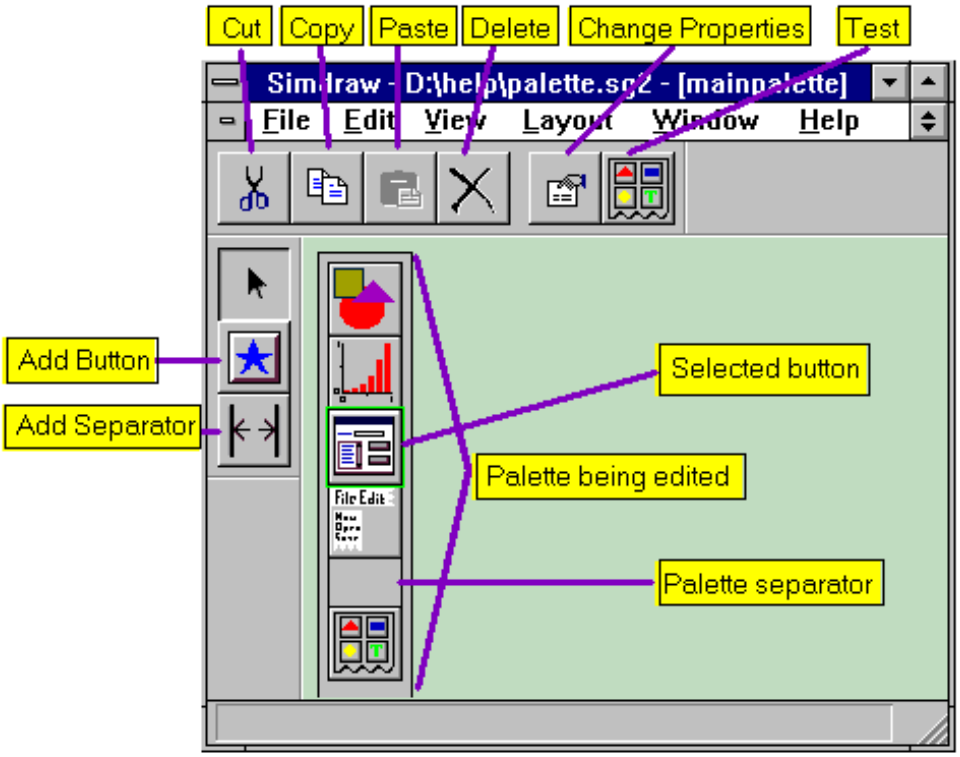


Figure 3-5. Palette Editor

3.17.1 Selecting and Moving (Rearrangement of) Buttons

A palette button or separator item can be selected by clicking the mouse button over the top of it. Selected buttons are marked by a bordering green or cyan box. Multiple items can be selected by holding down the **Shift** key and clicking on several items. To move a palette button from one place to another, drop it over the top of the button whose position you want it to occupy. You can select the palette and edit its properties, but it cannot be moved.

You are not allowed to resize palette buttons or the palette. All palette buttons are sized equally based on the size of the "first" button (at top left hand corner of the palette). This "first" palette button is automatically made big enough to contain its bitmap icon.

However, palette separators can be resized. Resizing a separator has the effect of adjusting the space between palette buttons. To resize the separator, first select it and then drag the green resize handle shown on a side of the selection rectangle.

3.17.2 Using the Clipboard (Cut, Copy and Paste)

The **Palette Editor** supports the standard cut, copy, and paste operations found under the **Edit** menu. The **Cut** option deletes selected items and places them in the clipboard. The deleted item remains on the clipboard until the next time you use the **Edit/Cut** or **Edit/Copy** options. You can use the **Edit/Paste** option to paste as many copies as you want from the clipboard onto any open menu pane. Items can be deleted without changing the clipboard by using the **Edit/Delete** option.

The clipboard is shared among all active **Palette Editor** sessions. You can copy graphics from one palette into another by activating the source edit window, using the **Copy** option, and then activating the destination editor and using the **Paste** option.

Note: The palette itself can never be cut, copied, or deleted. It can, however, be selected for the purpose of changing its properties.

3.17.3 Editing the Palette

A palette contains an array of selectable palette buttons. Palettes can be attached to any edge of the application window, or be floating (not unlike a modeless dialog box.) On MS Windows systems, palettes can be *dockable* meaning they can be moved from one edge of the window to another while running the application. Palettes cannot be resized; they are automatically sized to fit their contents. Double clicking on a palette will display the following detail:

- **Library Name** – The name of this palette in the graphics library.
- **Title** – Title text displayed in the header bar of a floating palette.
- **# Columns for Left/Right Dock** – Number of columns of palette buttons and separators whenever the palette is docked on the left or right edges of the window, or the palette is floating.
- **# Rows for Top/Bottom Dock** – Number of rows of palette buttons and separators whenever the palette is docked on the top or bottom edges of the window.
- **# Columns for Floating** – Number of columns of palette buttons and separators whenever the palette is not docked on a window edge, but floating free.

You can also define how **ACCEPT.F** will behave when displaying the palette. See paragraph 5.3.2.

3.17.4 Editing a Palette Button

Palettes are occupied by an array of palette buttons. A palette button has the following attributes which are adjustable via the **Edit/Properties** menu option:

- **Reference (Field) Name** – Any button added to the palette can be accessed from inside an application by specifying a **Reference** or **Field** name. The field name is passed to the callback routine whenever the button is clicked on.
- **Icon Name** – The name of the bitmap resource or file (without extension) icon displayed on the front of the palette button. Pressing the small browse "**..**" button next to this text box will allow you to browse the file system to select a bitmap file name. Remember that the bitmap file MUST be in the same directory as your library (**.sg2**) file.
- **Status Message** – Text displayed in pane 0 of the parent window's status bar (if present) whenever the pointer passes over this button.
- **Tool Tip** – Identifies the tool tip pop up message shown at the pointer's current location when it passes over this button.
- **Momentary/Draggable/Toggle** – Determines the variety of input interaction. One of three button types can be selected:
  1. **Momentary** – Button will automatically pop back up after it is pressed.
  2. **Toggle** – Two state button. The state (up or down) alternates with each activation.
  3. **Draggable** – Like **Toggle** but allows you to hold the mouse button down and drag an outline of the palette button bitmap onto the window.
- **Icon Button/Color Button** – If the **Icon Button** item is activated, the face of the palette button will show the bitmap defined by the **Icon Name** field above. For **Color Buttons** the button will be colored using the R,G,B parameters defined below.
- **Button Face Color (Red,Green,Blue)** – You can set the color of the **Color Button**s through these value boxes. Color is defined by the percentage of Red, Green, and Blue (range [0-100]).

3.17.5 Editing Palette Separators

Palette separators receive no user input and cannot be seen on the test palette. They only serve to provide a gap between buttons. This separation can be changed either by dragging the resize tag on a selected separator, or by using the **Edit/Properties** menu option. Separation is defined by percentage of button width (or height), and ranges from 0 to 100.

4. Creating Presentation Graphics

This chapter describes features of the SIMSCRIPT II.5 language which support both the display of numerical information in a variety of static and dynamic chart formats, and the representation of changing values using dynamic "smart icons." Graph types include:

- Histograms,
- Grouped histograms,
- Dynamic bar charts,
- Pie charts,
- X-Y plots, and
- Trace plots exhibiting variables traced over time.

These features are supported by SIMSCRIPT II.5 language enhancements. In general, data are collected with versions of the `TALLY` statement and `ACCUMULATE` statements. Data are then displayed with several forms of the `DISPLAY` statement.

Both static and dynamic graphs are supported. Data structures can be defined to represent either the immediate state of variables or to generate dynamic displays that automatically change over simulated time, as the program modifies the variables being observed. The dimensionality of structured data must match the dimensionality of the icon—for example, a scalar value can be shown on a dial or level gauge, but a bar-chart, graph, or piechart is required to represent an array of values. In general the type and format of the icons are selectable at run time without the need to recompile any code.

To create presentation graphics:

1. Declare the relevant globally-defined variables as `DISPLAY` in the program preamble;
2. Format the display icons using SIMDRAW and save them in **`graphics.sg2`**;
3. Add statements to the executable code to associate display variables with the icon description stored in **`graphics.sg2`**.

Each of these steps is described below.

4.1 Variable Declaration

Any globally defined numeric variables, either scalars, arrays, or attributes, may be declared as `DISPLAY` variables in the preamble with the statement:

```
DISPLAY VARIABLES INCLUDE statement  name1, name2....
```

This declaration is made *in addition* to normal variable declarations.

Histograms are a special case. Histogram names should not be included in a `DISPLAY VARIABLES INCLUDE...` statement. Histograms automatically acquire display capa-

bility. Including the special qualifier DYNAMIC creates histograms that are automatically updated *as the data change*, without the necessity of issuing repeated DISPLAY commands:

```
TALLY histname(low TO high BY interval) AS THE DYNAMIC HISTOGRAM OF name
```

(where **name** is already defined as an attribute or global variable).

4.2 Displaying Presentation Graphics

Once the icons are defined, a display variable is associated with an icon descriptor file using the statement:

```
DISPLAY [HISTOGRAM] name1,name2,.. WITH "iconname"
   [AT (posx, posy)]
```

or

```
SHOW name1, name2,... WITH "iconname" [AT (posx,posy)]
```

DISPLAY causes the icon to become immediately visible. SHOW merely associates the variable with the icon. A statement:

```
DISPLAY name
```

or assignment to the variable, in the case of dynamic icons, is then required to make the icon visible at some later point in execution. If more than one data set is to be displayed in a single chart, all should be named in a single SHOW/DISPLAY statement. The clause "[AT(posx,posy)]" is optional. Any positioning should be in NDC units (Normalized Device Coordinates, see paragraph 6.2) and overrides the icon position determined during icon editing. Subsequent DISPLAY statements for this variable should not include the WITH clause.

Histograms are again a special case. For histogram data, the icon association must be made by executing the statement:

```
SHOW <HISTOGRAM> histname WITH "iconname"
```

before any values are assigned to the monitored variable (and before any other reference is made to the histogram name). This is because the stored information is maintained in the histogram structure. The first assignment to the monitored variable for dynamic histograms will cause the histogram to be displayed, and any further references will cause the display to be updated. Thus, for a dynamic histogram, DISPLAY statements are redundant.

If variable names are used for the histogram limits (low, high, interval) these will be automatically initialized from the X-axis graduations specified on the graph icon. These can be edited in SIMDRAW. Should the displayed bounds on the Y-axis be exceeded during the simulation, the histogram will rescale automatically.

Icons may be erased by specifying their display variables in an ERASE statement .

```
ERASE name1, ...
```

Dynamic histograms may be destroyed by specifying their names in an ERASE HISTOGRAM statement:

```
ERASE HISTOGRAM name1, ...
```

4.3 Examples

4.3.1 Example 1: A Simple Tallied Histogram

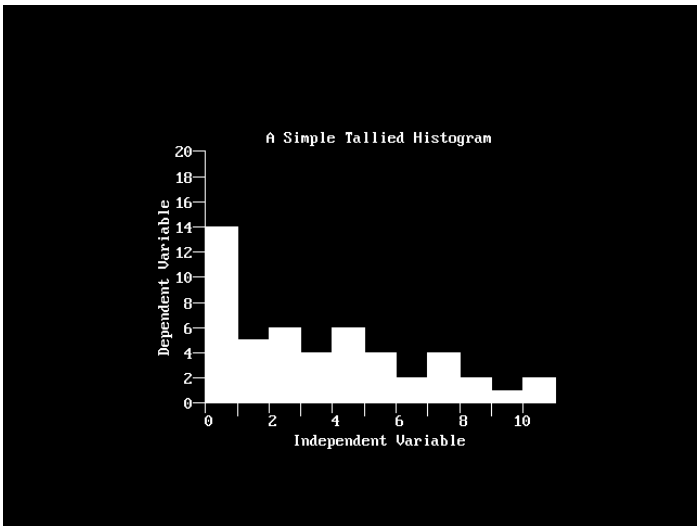


Figure 4-1. Example 1

```
Preamble
   define RANDVAR as a double variable
   tally HISTO(0 to 10 by 1) as the histogram of RANDVAR
end
main
   define COUNT, NSAMPLES as integer variables
   show HISTO with "hist.grf"                          '' *
   let NSAMPLES = 50
   for COUNT = 1 to NSAMPLES
         let RANDVAR = exponential.f(5.0,1)
   display HISTO                                       '' *
   read as /
end
```

This program is conventional SIMSCRIPT with the exception of the two lines commented with `'' *`.

Be sure that the `show. . .` statement precedes the first assignment to **RANDVAR**. This assignment triggers data collection, and the icon structure must be known by then. The `display. . .` statement makes the icon visible. The final **`read as /`** is just waiting for an **Enter** key (carriage return) before terminating the program, and thus erasing the graphics display.

4.3.2 Example 2: A Time-Weighted Accumulated Dynamic Histogram

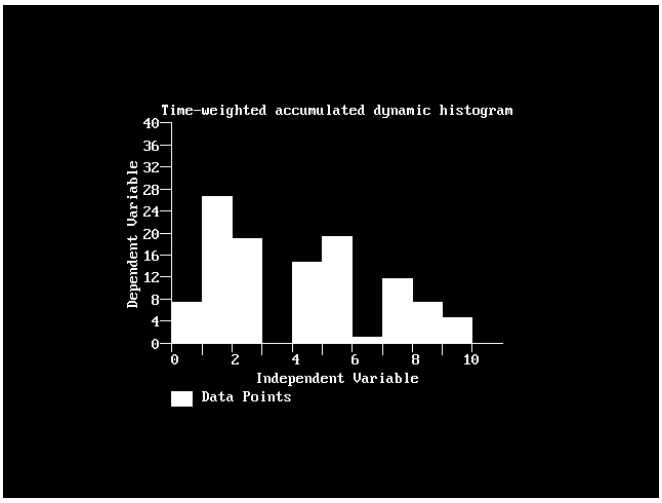


Figure 4-2. Example 2

```
Preamble
   define RANDVAR, LO, HI, INTERVAL as double variables
   accumulate HIST(LO to HI by INTERVAL) as the dynamic histogram of
RANDVAR
   processes include SAMPLE
end
main
   show HIST with "hist.grf"
   activate a SAMPLE now
   start simulation
   read as /
end
process SAMPLE
   until TIME.V gt 100
   do
       wait exponential.f(5.0, 1) units
       let RANDVAR = uniform.f(0, 10, 2)
   loop
end
```

Accumulated statistics are weighted by the duration of simulated time for which the value remains unchanged. For this reason the example is written to use a process to generate the sample data, waiting for simulation time to elapse between each sample.

Note: Histogram limits are declared in terms of variables. These variables obtain their values from the X-axis specification of the graph icon.

Because the histogram is specified as dynamic, it redisplays any time the variable **RANDVAR** is assigned a value. A DISPLAY statement would be redundant. Substituting the word **display** for **show** would cause the graph to display immediately upon loading, before the simulation has started.

4.3.3 Example 3: Displaying Simple Scalar Values

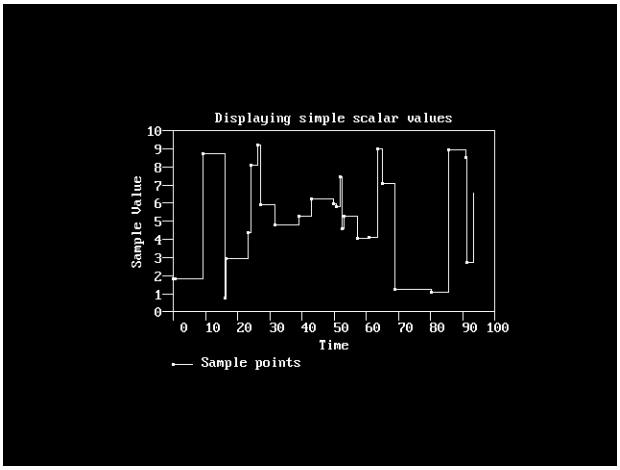


Figure 4-3. Example 3

```
Preamble
   define RANDVAR as a double variable
   display variables include RANDVAR
   processes include SAMPLE
end
main
   show RANDVAR with "trace.grf"
   activate a SAMPLE now
   let TIMESCALE.V = 10
   start simulation
   read as /
end
process SAMPLE
   until TIME.V gt 100
   do
         let RANDVAR = uniform.f(0, 10, 2)
         wait exponential.f(5.0, 1) units
   loop
end
```

This example is also constructed as a simulation so that it can be used to illustrate the use of a trace plot. A dial or level meter could be substituted merely by editing a suitable icon and naming it **`trace.grf`**.

This example is more interesting if simulation time is scaled to real time so that the process actually waits some noticeable time between samples. Use the global system variable **`TIMESCALE.V`** to achieve this. **`TIMESCALE.V`** specifies the number of hundredths of a realtime second that should correspond to one unit of simulated time.

4.3.4 Example 4: Using a Trace to Plot X-Y Curves

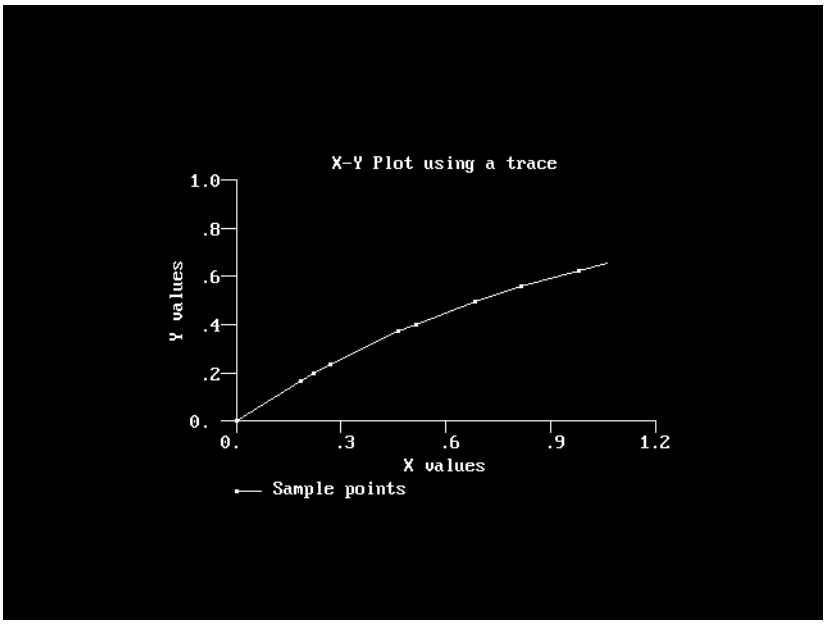


Figure 4.4 Example 4

```
Preamble
   define X, Y as a 1-dim double array
   define YPLOT as a double variable
   display variables include YPLOT
end
main
   define SAVTIME as a double variable
   reserve X(*), Y(*) as 10
   for I = 1 to 9
         let X(I+1) = X(I) + uniform.f(0, 0.2, 1)
   for I = 1 to 10
         let Y(I) = 1 - EXP.C ** (-X(I))
   show YPLOT with "trace.grf"
   let SAVTIME = TIME.V
   for I = 1 to DIM.F(X(*))
         do
             let TIME.V = X(I)
             let YPLOT = Y(I)
         loop
   let TIME.V = SAVTIME
   read as /
end
```

This example shows how a trace plot can be "tricked" into plotting one set of values, Y, against another set, X.

YPLOT is a display variable used solely to generate the trace plot.

Trace plots expect to derive the X-axis coordinate from the current value of **TIME.V**, the simulation time. If using this trick in the context of a simulation, be sure to save and restore **TIME.V**.

4.3.5 Example 5: The Bank Model

As an example of the use of presentation graphics in a simulation model, a very simple single-queue, multiple-server bank model has been augmented to include displays of the simulated time on an analog clock, the queue length as a level meter, and the waiting time of customers as a dynamic bar chart (or histogram). All code which is not essential to the illustration of graphical concepts (i.e., the menu-driven selection of input parameters and the gathering and reporting of numerical statistics) has been omitted. The code and icon files for the complete model are included on the distribution kit for SIMSCRIPT II.5. Part of the output is shown in figure 4.5.

The simulation model is described in the `Preamble, Main`, the `INITIALIZE` routine, and in the `GENERATOR` and `CUSTOMER` processes. The presentation graphics are described in lines 12 to 17 of the preamble, line 3 of the main routine, and in two routines, **`CLOCK.UPDATE`** and **`INITIALIZE.GRAPHICS`**.

In addition to the model enhancements, SIMDRAW was used to produce descriptions of three icons to be used for display purposes. These are: **`clock.grf, queue.grf`**, and **`wait.grf`**.

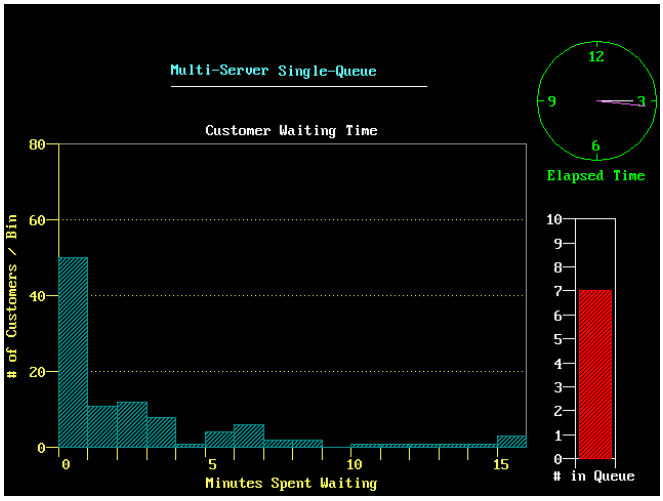


Figure 4-5. The Bank Model

```
preamble                ''  BANK  - Modernizing a Bank
    normally, mode is undefined

    processes include GENERATOR and CUSTOMER

    resources include TELLER

    define NO.OF.TELLERS as an integer variable
    define MEAN.INTERARRIVAL.TIME, MEAN.SERVICE.TIME, DAY.LENGTH
       and WAITING.TIME as real variables

     Define WLO, WHI and WDELTA as integer variables
    Define CLOCKTIME as a double variable
```

```
   Display variables include CLOCKTIME, N.Q.TELLER

   Graphic entities include SHAPE

   Tally WAITING.TIME.HISTOGRAM (WLO to WHI by WDELTA)
      as the dynamic histogram of WAITING.TIME

 end  ''preamble

main
   call INITIALIZE
   call INITIALIZE.GRAPHICS
   start simulation
   read as /
end  ''main

routine INITIALIZE
   let NO.OF.TELLERS = 4
   let MEAN.INTERARRIVAL.TIME = 2.0
   let MEAN.SERVICE.TIME = 7.0
   let DAY.LENGTH = 4 / hours.v    '' days
   create every TELLER(1)
   let u.TELLER(1) = NO.OF.TELLERS
   activate a GENERATOR now
end  '' routine INITIALIZE

routine INITIALIZE.GRAPHICS

   Define DEVICE.ID and TITLE as pointer variables

   Let timescale.v = 1000  '' clock ticks (1/100 sec) per unit
   Let timesync.v = 'CLOCK.UPDATE'

   Display CLOCKTIME with "clock.grf"
   create a SHAPE called TITLE
   display TITLE with "title" at (15000.0, 21000.0)

   Display N.Q.TELLER(1) with "queue.grf"
   Display histogram WAITING.TIME.HISTOGRAM with "wait.grf"

end  ''INITIALIZE.GRAPHICS

routine CLOCK.UPDATE given TIME yielding NEWTIME
    Define TIME, NEWTIME as double variables
    Let NEWTIME = TIME
    Let CLOCKTIME = TIME
End  ''CLOCK.UPDATE

process GENERATOR
   until time.v >= DAY.LENGTH
   do
      activate a CUSTOMER now
      wait exponential.f(MEAN.INTERARRIVAL.TIME, 1) minutes
   loop
end  ''GENERATOR

process CUSTOMER
   define ARRIVAL.TIME as a real variable
   let ARRIVAL.TIME = time.v
   request 1 TELLER(1)
   let WAITING.TIME = (time.v - ARRIVAL.TIME) * minutes.v * hours.v
   work exponential.f(MEAN.SERVICE.TIME, 2) minutes
   relinquish 1 TELLER(1)
end  ''CUSTOMER
```

5. Forms and Graphical Interaction

5.1 Introduction

A graphical user-interface is made up of menu bars, buttons, pickable icons and other constructs, each with its own behavior and function in the program. SIMGRAPHICS II incorporates a form-based approach for easily creating these interfaces and adding them to programs. This chapter is organized into three loose parts:

- After reading paragraphs 5.2 through 5.4 you should be able to easily use interactive graphics to improve your programs. These paragraphs give all the information needed for simple input using powerful SIMGRAPHICS II entities.
- Paragraphs 5.5 and 5.6 are meant as a more complete description of the basic SIMGRAPHICS II forms-based concepts and commands.
- Refer to paragraphs 5.7 through 5.9 for complete details of each SIMGRAPHICS II entity (Button, Menu bar, …) as they are needed. Paragraph 5.10 contains several simple example programs, with relevant concepts listed before each. These should be referred to if the use of any concept seems unclear.

Constructing an application interface involves designing the layouts of the interactive graphics and writing the code to manage their behavior. Without SIMGRAPHICS II, the application developer is often burdened with controlling, at a detailed level, the mechanics of the particular interaction environment: Has the mouse button been pressed?; Was that a backspace key?; Should I echo it?; etc.

SIMGRAPHICS II offers tools to help design the layout of the interaction screens. These may be composed by combining a number of interaction primitives to display lists, accept numeric values, check options, etc. These layouts are then filed for later use by an application program. At execution time, SIMGRAPHICS II takes over most of the interaction management—tracking the mouse, toggling highlights, scrolling lists. This leaves the application developer free to concentrate on the mechanics of his or her program.

A form is composed of a group of fields. There are three principle types of forms: pull-down menus, dialog boxes and palettes that may contain values boxes, text boxes, list boxes and buttons.

Forms are generated and modified by SIMDRAW. This editor was created using the forms concepts, so it is an excellent example of its own capabilities. The editor saves coded descriptions of these structures to files. The application program can retrieve any form and rebuild its structure, with a single statement. A function call initiates and manages the interactive behavior of a form. The run-time support library knows how to generate the display screens and manage the details of the user interaction. The steps for accomplishing the simplest interactions are summarized as:

1. Design a form with SIMDRAW.

2. Save the form.
3. Use the command SHOW and the function **ACCEPT.F** in your program to display and input information with your form.
4. Use the built-in SIMSCRIPT functions and attributes to identify fields and retrieve data.

During forms editing, each field is tagged with an arbitrary reference name. The application program uses this reference name to refer to a particular field within a form structure. This is how the program can retrieve data values and selections from the interaction and even alter the values or other internals in some of the fields. Each field has control attributes, specified during forms editing, which help determine its behavior during the interaction.

The application program has the option of selectively using code to manage the behavior of specific fields in a form. This is done by supplying a control routine for a form. This control routine is invoked each time any field in the form is selected. Based on the field Identifier, the control routine can supply any behavior, analysis, or validation desired.

5.2 Creating a Form

You create a form with SIMDRAW. Typically, this is run in the subdirectory of the model for which the form is being created, because, by default, files are saved in this subdirectory.

Types of fields available in SIMDRAW include buttons, radio buttons, check boxes, text boxes, value boxes, list boxes, combo boxes, tables, tree lists and a progress bar. Refer to table 5-1.

Different field types may have different associated data types. The attributes **DDVAL.A, DTVAL.A**, and **DARY.A** are used to access numeric, string, or array values respectively, given the pointer to the relevant field.

Table 5-1. Simple Behaviors and Field Types

In the first two columns of the table below are the requirements you may have for a particular field in a dialog box. The third column lists which attributes of the field pointer (obtained by **DFIELD.F**) should be specified in your program. The fourth column lists all of the objects available from SIMDRAW that could be used to implement the requirement. These objects are described in greater detail later.

REQUIREMENTS	USER ACCESS	ENTITY ATTRIBUTES	CONTROLS TO USE
One numerical value	RW	DDVAL.A	Value box
One numerical value	R	DDVAL.A	Label Progress bar
One text value	RW	DTVAL.A	Text box
One text value	R	DTVAL.A	Label
One boolean value (0 or 1)	RW	DDVAL.A	Check box Radio button Menu item
Many text values	RW	DARY.A	Multi-line text box
Select from an array of text	R	DARY.A DTVAL.A DDVAL.A	List box Combo box Tree list Table

5.2.1 Reference Names and Field Attributes

Each field within a form, to be accessible to an application, must be tagged with a unique reference name. This name is a text string specified for the field during forms editing. The name is used when invoking a SIMSCRIPT run-time library function which accesses a specific field in the form data structure.

The attributes of any field are categorized as graphic, controlling its appearance, or control, determining its behavior. Values for these attributes may be chosen at editing time. The most important ones for now are:

- **The Reference Name.** This is the name you give a field when you create it in SIMDRAW. It is needed to refer to the field later.
- **Value Attributes.** These include **DDVAL.A, DTVAL.A**, and **DARY.A**, for numeric, text, and array values, respectively. These can be used for input or can be set by the program for use as output.

All of these can be set (or left to the default) in SIMDRAW. They will be elaborated on in later sections.

5.3 Using the Form in a Program

An application program refers to a form by either a local or a global pointer variable:

```
define FORM.PTR as pointer variable
```

Then, a SHOW statement,

```
show FORM.PTR with "formname"
```

rebuilds the internal data structure for the form from the specified form file. **FORM.PTR** now points to the form.

Note that the SHOW statement does not display the form. It loads it from the form file and assigns it to the pointer variable. The form will be automatically displayed when input is needed.

5.3.1 Using ACCEPT.F

The display interaction is initiated by invoking a run-time support function **ACCEPT.F**, which takes as parameters the pointer to the form and, optionally, an associated control routine. This function displays the form and manages the interaction dialog. The function call will not be completed until either a terminating button (i.e. a button marked as "Terminating" in SIMDRAW) is selected, or until forced to do so by the control routine. When completed, the reference name of the last selected field is returned.

```
define FIELD.NAME as a text variable
 . . .
let FIELD.NAME =ACCEPT.F(FORM.PTR, control.routine)
            [or ACCEPT.F(FORM.PTR, 0)]
```

By examining the returned value, the application can make a decision about the outcome of the interaction. For example, a very simple form might present only two selectable fields with Identifiers GO and STOP. The returned value indicates whether the user wants to proceed past this point in execution.

```
if ACCEPT.F(FORM.PTR, 0) eq "STOP"
    stop
endif
```

Typically, a form will include two terminating buttons, (labelled, for example, **Ok** and **Cancel**) which are used to indicate that the user wishes to terminate the input. **Ok** tells the program to accept the data or instructions gained in the interaction. **Cancel** instructs the program to back up a step, causing the interaction to have no effect.

If data is to be retrieved, selected fields may be examined, using a run-time support function, **DFIELD.F**. This function takes as input the field Identifier tag assigned during forms construction, and the pointer to the form.

```
let ELEVATION.PTR = DFIELD.F("elevation", FORM.PTR)
```

Data is retrieved using an attribute which depends on the field type. These attributes include **DDVAL.A, DTVAL.A**, and **DARY.A**., for numeric, text, and array values respectively. For instance:

```
let ELEVATION.VARIABLE = DDVAL.A(ELEVATION.PTR)
```

Table 5-2. Some Graphics Interaction Constructs

SHOW:	Loads the form and sets a pointer to it. `SHOW FORM.PTR with "Filename"`

ACCEPT.F:	Initiates a graphics interaction.
	`let EXIT.STATUS = ACCEPT.F(FORM.PTR, CONTROL.ROUTINE)`
	`CONTROL.ROUTINE may be 0`

DFIELD.F:	Returns the pointer to a field given its name and the form pointer. `let ELEVFIELD.PTR = DFIELD.F("elevation", FORM.PTR)`
DDVAL.A:	Accesses a numeric value associated with a field. `let ELEV.VARIABLE = DDVAL.A(ELEVFIELD.PTR)`
DTVAL.A:	Accesses a text value associated with a field. See DDVAL.A. `let PILOT.NAME = DTVAL.A(NAMEFIELD.PTR)`
DARY.A:	Accesses an array pointer associated with a field. See DDVAL.A `let ENGINE.THRUSTS(*) = DARY.A(THRUSTFIELD.PTR)`

5.3.2 Interaction Modes

With regard to execution control, the behavior of **ACCEPT.F** can be defined using the **Edit/ Properties..** option on the dialog box, menu bar, or palette from within SIMDRAW. The **ACCEPT.F** function will behave according to one of three *interaction modes*:

1. **Asynchronous**: If this interaction mode is used, **ACCEPT.F** will suspend the active process when called. Whenever a status value of "1" is returned from the control routine or a terminating button is pushed, this process is resumed. If there is no simulation running and hence no active process, the **Synchronous** interaction mode is used.
2. **Synchronous**: Regardless of the simulation, **ACCEPT.F** will not return until a status value of "1" is returned from the control routine or a terminating button is pushed.

3. **Don't wait**: **ACCEPT.F** will not wait for any action by the user but will return immediately. Subsequent action on the form will invoke the control routine.

5.4 Field Attributes

All attributes, including value attributes, can be set within SIMDRAW.

5.4.1 Value Attributes

Each field is represented by a SIMSCRIPT structure. Each of these field structures has several value attributes that are accessible from application code.

Having the reference name and the form pointer, a run-time support function will return a pointer to the field structure:

```
define FIELD.ptr as pointer variable
let FIELD.PTR = DFIELD.F(FIELD.NAME, FORM.PTR)
```

Field attributes accessible to the application code are defined as:

```
define DDVAL.A as double variable
define DTVAL.A as text variable
define DARY.A  as pointer variable
```

All numeric values are held in **DDVAL.A**. This includes the values 0 and 1 associated with binary selection of options.

Text data values are held in **DTVAL.A**.

DARY.A is commonly used as a pointer to a one dimensional array of text variables when dealing with selections from lists.

These can be accessed by a program as in the following examples:

```
let I = DDVAL.A(FIELD.PTR)
let DTVAL.A(DFIELD.F(FIELD.NAME, FORM.PTR)) = TEXT.VARIABLE
let TEXT.ARRAY(*) = DARY.A(FIELD.PTR)
```

Field attributes are initialized to zero for numbers and pointers and to null string for text variables. A program can get and set the values of these attributes at any time after loading the form and before destroying it.

If **DFIELD.F** cannot find a field with a reference name matching the specified **FIELD.NAME,** it returns a null pointer, i.e. zero. So, if there is a chance that the Identifier will not be matched, program as follows:

```
let FIELD.PTR = DFIELD.F(FIELD NAME, FORM.PTR)
if FIELD.PTR ne 0
  let PART.NUMBER = DDVAL.A(FIELD.PTR)
else
 '' error processing
endif
```

5.4.2 Terminating Buttons

Typically, a form has more than one interactive field. These may be selected in any order, and may be selected repeatedly during a single form interaction. At least one button, however, should be designated from within SIMDRAW as a termination button, often labelled **OK**, **ACCEPT**, or **RETURN**. This terminates the display management phase and returns to the application, which can then interrogate the form structure to retrieve the data values from each field. As previously noted, a form may have more than one terminating field, e.g. **OK** and **CANCEL**. Thus, selection of the terminating field may also convey some information in addition to ending the dialog.

5.4.3 Verifying Buttons

From within SIMDRAW, you can mark a button as "Verifying". Typically, the **Ok** button is marked as a verifying button. Clicking on a verifying button causes all ranges on value boxes to be checked.

5.5 Form Control Routines

By default, **ACCEPT.F** calls no control routine. This is specified by passing 0 as the second argument. In this case, only the standard processing is called. The automatic processing can accept or reject any user input (for instance if a number is out of bounds), and will take care of terminating the dialog interaction, when a terminating field is clicked.

Additional processing can be specified by declaring a control routine and passing its name as the second argument to **ACCEPT.F**. A control routine is declared as:

```
routine DIALOG.ROUTINE
  given FIELD.NAME, FORM.PTR
  yielding FIELD.STATUS

define FIELD.NAME as       text variable
define FORM.PTR as         pointer variable
define FIELD.STATUS as     integer variable
```

This control routine is called after user input to any field on the form, after the automatic processing is completed.

The control routine is passed the field identifier. Based on this it can select suitable validation or cross-checking to perform. The return value of **FIELD.STATUS** will communicate the following to **ACCEPT.F**:

-1 Reject the input (and retry)

0 Accept the input

1 Terminate the interaction.

Setting the return value of **FIELD.STATUS** affects the continuing interaction.

Control routines, if present, are also called with the predefined reference names `INITIALIZE` and `BACKGROUND`. The control routine is called with the field name `INITIALIZE` only once, before the dialog box is displayed. It is called with the field

name BACKGROUND whenever the user clicks on the canvas of the graphics window. You can retrieve the location of this mouse click through the **LOCATION.A** attribute of the display entity pointer **DINPUT.V** (if nonzero).

5.6 Details of Field Operations

The following operations can be applied to individual fields or to an entire form.

5.6.1 The DISPLAY Command

To redisplay a form:

```
display FORM.PTR
```

If some initially displayed fields were erased, they will be redisplayed.

To redisplay a field in its current position:

```
display FIELD.PTR
```

Once the form has been displayed, individual fields can displayed and redisplayed. The values displayed in the fields are taken from the appropriate field attributes, depending on the field type.

Displaying a radio button has a side effect when **DDVAL.A = 1**. If another radio button in the same group has **DDVAL.A = 1**, it is set to zero and that button is redisplayed.

5.6.2 The ACCEPT.F Function

```
let FIELD.ID = ACCEPT.F(FORM.PTR, 0)
let FIELD.ID = ACCEPT.F(FORM.PTR, 'DIALOG.ROUTINE')
```

We have already described the use of **ACCEPT.F** for initiating a form-driven dialog.

```
let FIELD.ID = ACCEPT.F(FIELD.PTR, 0)
```

ACCEPT.F called on a form returns the reference name of the field that caused the termination of the dialog. **ACCEPT.F** should not be called on a single field.

5.6.3 The ERASE Command

```
erase FORM.PTR
```

This causes the form to disappear from the screen. A DISPLAY statement, or another **ACCEPT.F**, specifying **FORM.PTR** will bring it back. Note that any values in the form are unchanged.

```
erase FIELD.PTR
```

This erases a single field from the screen.

5.6.4 The DESTROY Command

The destroy command not only erases a form from the screen, but clears all memory allocated to it, reversing the effect of the SHOW command. The syntax is:

```
destroy FORM.PTR
```

5.6.5 The SET.ACTIVATION.R Routine

This routine can be called to either deactivate (gray out) or activate (make usable) a field on the form. The syntax is:

```
call SET.ACTIVATION.R given FIELD.PTR, ACTIVATION.STATUS
```

The **FIELD.PTR** parameter is the field pointer obtained from **DFIELD.F**. **ACTIVATION.STATUS** is a flag where "0" means to gray out the field and "1" means to make it active. You can deactivate an entire form with **SET.ACTIVATION.R** if you pass the form pointer as its first parameter.

5.7 Dialog Boxes and Their Fields

5.7.1 Dialog Box

This is a container for all the other control types. You can set the initial screen position of the dialog box from within SIMDRAW by selecting the dialog box's background and using the **Edit/Properties...** option. The position can also be set programmatically using the **LOCATION.A** attribute of the form pointer. (The lower left-hand corner of the screen maps to coordinate (0,0) while the upper right-hand corner maps to (32767,32767). Use SIMDRAW to define which of the four corners of the dialog box is positioned. Note that the value of **LOCATION.A** is updated automatically whenever the user repositions the dialog.

A dialog box can be either **tabbed** or **non-tabbed**. A tabbed dialog box contains a section of overlapping pages of fields. Generally, fields in the same page should have a common functionality. A page is made visible when the user clicks on its **tab** at the top of the page. A tab is composed of a text label and optionally a small bitmap icon. Tabbed dialog boxes are created within SIMDRAW and require no additional programming over traditional dialog boxes. The control routine is NOT called when the user selects a tab page. Management of these pages is done automatically.

Button

A Button is a box with an explanatory text string in it. It is the simplest input field. When a button is clicked on, the control routine gets only the field name of the field. It gets no other data. Remember that buttons can be "terminating" or "verifying".

Check Box

A Check Box toggles between the values zero and one. When the form is loaded, check boxes have the default value set during forms editing. The state of a check box is determined by setting or examining the attribute **DDVAL.A** of that check box.

When a check box is selected, **ACCEPT.F** toggles the value of **DDVAL.A**, redisplays the check box to reflect the new value, and calls the control routine, if provided. The field is not redisplayed after returning from the user's control routine.

Radio Buttons

Radio Buttons are differentiated from check boxes in that they are contained in a radio box. Only one radio button in the box can be turned on. The state of a radio button is determined by the attribute **DDVAL.A**. It assumes values zero and one. When the form is loaded, the radio buttons have the values set during forms editing. The radio box itself has no value attributes.

When a radio button is selected, **ACCEPT.F** sets the value of **DDVAL.A** to one, redisplays the field to reflect the new value, and calls the control routine, if one is provided.

Text Box (Editable)

The value associated with a Text Box is a text string in **DTVAL.A**. The initial value can be specified when creating the field by using SIMDRAW. If the field is marked as **Selectable using Return** in SIMDRAW, the control routine will be called if the user presses **Return** after entering the data.

Text Label (Non-editable)

A label is used to place explanatory text, values or titles in a dialog box. The text of the label can be reset programmatically but cannot be modified by the user. The control routine is not called when the user clicks on a label. From within SIMDRAW you can define whether the text is specified by the **DTVAL.A** or the **DDVAL.A** attribute of its field pointer. SIMDRAW can define one of the following behaviors for a text label:

a. Use the **DTVAL.A** attribute to define the text. There is no limit to string length.
b. Use the **DTVAL.A** attribute to define the text. Allocate a fixed number of places for the string.
c. Use the **DDVAL.A** attribute to define a real value displayed by the label. The total number of places, and number of places after the decimal point can be defined in SIMDRAW.

For example, if the label's reference name is "MY.LABEL", you can programmatically set the label as follows:

```
Let DTVAL.A(DFIELD.F("MY.LABEL", FORM.PTR)) = "Hello World"
```

or

```
Let DDVAL.A(DFIELD.F("MY.LABEL", FORM.PTR)) = 12.5
```

Multi-line Text Box

A multi-line text box allows the user to type in as many lines of text as he wants to. Horizontal and vertical scroll bars are shown if the visible area of the text edit window is not large enough to enclose the text. The initial text can be set from within SIMDRAW or by program code. The current contents of the text box are held by the **DARY.A** attribute which points to a SIMSCRIPT II.5 array of text. Note that the control routine is NOT called when the **Return** key is pressed. The **DARY.A** attribute is automatically updated to reflect the current text box contents before the control routine is called in response to an action on any *other* field in the form.

Value Box

The Value Box is used to read a number. When it is selected the box prompts for input. The value of the number is in **DDVAL.A** and is initially zero. In SIMDRAW this field can be associated with a range of acceptable values by specifying the minimum and maximum value. The number read from the field is not accepted if it falls outside of this range. The maximum and minimum values can only be set or examined during dialog box editing. The value box's contents is checked when a verifying button is clicked on.

Progress Bar

A Progress Bar is useful for indicating the “time to completion” of some task being performed by your program. It is composed of a horizontal bar whose size indicates a magnitude relative to some lower and upper bound. The minimum and maximum values are defined in SIMDRAW while the length of the bar is defined programmatically, using the **DDVAL.A** attribute. The user cannot interactively change the position of the bar with the mouse.

List Box

A List Box is associated with an array of text variables. Each array element is displayed on a separate line of the list box. To select an item from the list, the mouse is clicked over that line. New items can not be typed into a list box. Only existing items can be selected. To add new items to the list box, a program must add them to the list box text array. The array pointer **DARY.A** points to the array of text variables that represent the list. This pointer is initially zero. The index of the selected element is in **DDVAL.A**. The selected array element is copied into **DTVAL.A**.

If the array has more elements than the list box can display, a vertical scroll bar will be shown on the right-hand side. By clicking over the up- or down-arrow in the scroll bar, the additional list items can be brought into view. This scrolling is done automatically. The control routine is called only when a selection is made.

When an item from the list box is selected, **ACCEPT.F** updates **DDVAL.A** and **DTVAL.A**. The selected item is highlighted and highlighting is removed from the item that was previously selected.

List boxes can be defined in SIMDRAW to allow multiple selections. The callback routine will be called each time an item is selected. The **DDVAL.A** attribute of the list box field will

contain the index of the item last selected. To synchronously poll a list box for its selected contents the **LISTBOX.SELECTED.R** routine is used. This routine can also be used to determine if the user has **double-clicked** on an item in the listbox. Given a pointer to the list box field and item number, this routine yields 2 if the item has been double-clicked on, 1 if this item is selected, and 0 otherwise.

```
  ''-- Synchronous polling of multiple selection list
  ''-- boxes.  First get input from the form
let FIELD.ID = ACCEPT.F(FORM.PTR, 0)
let LIST.ITEMS(*) = DARY.A(DFIELD.F("LISTBOX", FORM.PTR))

  ''-- use a loop to show which items were selected
for I = 1 to DIM.F(LIST.ITEMS(*))
do
    call LISTBOX.SELECTED.R
            given DFIELD.F("LISTBOX", FORM.PTR), I
            yielding ITEM.SELECTED.FLAG
      select case ITEM.SELECTED.FLAG
         case 0
            write LIST.ITEMS(I) as "ITEM ", T *, " not selected.", /
         case 1
            write LIST.ITEMS(I) as "ITEM ", T *, " was selected!", /
         case 2
            write LIST.ITEMS(I) as "ITEM ", T *, " was double-clicked!", /
      endselect
loop
```

Note: To deselect all items in a multiple selection list box, set its **DDVAL.A** attribute to 0 and redisplay the field.

```
''-- deselecting all selected items in a multiple selection list box
let DDVAL.A(DFIELD.F("LISTBOX", FORM.PTR)) = 0
display DFIELD.F("LISTBOX", FORM.PTR)
```

Combo Box

A combo box is made up of a text box and an initially hidden list box which is displayed when the dropdown button is pushed. **DARY.A** contains a selectable list of choices for the text displayed in the text box. When the user picks one of these text strings, it is automatically displayed in the box. If a combo box is defined to be "Editable" from within SIMDRAW, the user is allowed to type his own choice into the text box (instead of picking from the list). The control routine will be invoked whenever the user selects an item in the list. If the combo box is defined as **Selectable using Return** in SIMDRAW, then the control routine is invoked when the user presses the **Return** key.

Tree View List

Lists of items can be viewed hierarchically with items containing other items. In addition, items in the list can be denoted with a small bitmap icon. Items can be added to the list either from within SIMDRAW or from your program.

The items contained in the list are defined through the **DARY.A** attribute. The name specification uses the "/" character to separate the container name from the item name and works much the same way as a path name for the file system, i.e.

```
<top_container_name>/.../bottom_container_name>/<item_name>
```

For example, if you wanted to show following items:

a. "San Fransisco" contained in "California" which is in "Usa States"
b. "San Diego" contained in "California" which is in "Usa States"
c. "Las Vegas" contained in "Nevada" which is in "Usa States"
d. "Berlin" contained in "Germany" which is in "Europe".

The **DARY.A** attributes should contain the following text strings:

```
"Usa States/California/San Fransisco"
"Usa States/California/San Diego"
"Usa States/Nevada/Las Vegas"
"Europe/Germany/Berlin"
```

To define the bitmap icon to place next to the item name, use the "|" character after the name specification to separate the path name from bitmap file (or resource) name (without extension). For example, to use bitmap ZOOM_L for item "Zoom Tool" in category "Tools", the array item would be:

```
"Tools/Zoom Tool|ZOOM_L"
```

To define an icon for a category, list the category by itself on a separate line with the "|" character and the bitmap names, i.e.

```
"Tools|SELTOOL_L"
```

If you need to use the "|" or "/" characters in your item name, you can literalize them using a preceeding backslash "\" character.

The **DDVAL.A** attribute of a tree list field pointer will contain the index of the last selected item. In the above example, if the user clicked on the "Las Vegas" item, the field's **DDVAL.A** attribute would be "3". You can set the selected item in the tree by setting **DDVAL.A** and redisplaying the field. From above, set the selected item to "San Diego":

```
...
let DDVAL.A(DFIELD.F("MY_TREE", DIALOG.PTR)) = 2
display DFIELD.F("MY_TREE", DIALOG.PTR)
...
```

Table

A table contains a matrix of selectable rectangular text cells. A 1-dimensional text array containing labels displayed in cells is accessible through **DARY.A**. This array can be set up programmatically, but it must contain the same number of text strings as total cells in the table. **DARY.A** is organized in row-major order. **DDVAL.A** contains the index of the item last selected, while **DTVAL.A** has the text for this item.

Setting the text for some row and column programmatically would involve statements of the form :

```
let ITEMS = DARY.A(TABLE.PTR)
let ITEMS((ROW-1) * NUMBER.COLUMNS + COLUMN) = TEXT.VALUE
display TABLE.PTR
```

Getting which column and row were selected can be done from within the control routine as follows:

```
let SELECTED.ROW =  DIV.F(DDVAL.A(TABLE.PTR)-1, NUM.COLUMNS) + 1
let SELECTED.COLUMN = MOD.F(DDVAL.A(TABLE.PTR)-1, NUM.COLUMNS) + 1
```

where row number one (1) is the top row in the table and column number one is the leftmost column.

In SIMDRAW, you can add row and column headers to your table. Row headers are shown in a column on the left-hand side, while column headers are laid out as the top row of the table. If headings are added to the table, the row headers become *column* number one, and the column headers become *row* number one.

5.8 Predefined Dialog Boxes

5.8.1 Standard Message Dialog

If you wish to display a simple, one line message to the user and force him to respond to this message before execution of the program can resume, you can use the **MESSAGEBOX.R** routine. This routine will instruct the toolkit to display a dialog box containing the one line message, and one button labeled **OK** or **Continue**. **MESSAGEBOX.R** will not return until the user presses this button.

```
let TITLE = "Completion Status..."
let MESSAGE = "Your task has been completed!"
call MESSAGEBOX.R given MESSAGE, TITLE
```

5.8.2 Custom Message Dialogs (Alert, Stop, Information and Question)

You can define more customized dialogs to deliver simple information to the user and receive a response. Your program can display **Alert, Question, Information** and **Stop** dialog boxes. These forms are built using SIMDRAW (using the **Message Dialog** button on the left-hand **Mode** palette shown with SIMDRAW's main window.)

A custom message dialog can be one of the following five styles:

a. Plain
b. Stop Sign
c. Question mark
d. Alert (Exclamation point)
e. Information.

It will contain one of the following sets of response buttons:

a. **OK** button only

b. **OK** and **Cancel** buttons

c. **Yes** and **No** buttons

d. **Yes, No** and **Cancel** buttons

e. **Retry** and **Cancel** buttons

f. **Abort, Retry** and **Ignore** buttons.

Custom message dialogs are displayed using the SHOW statement and then the **ACCEPT.F** routine (like a conventional dialog). Control routines are not used with message dialogs. The text of the message can be set from SIMDRAW or by your program. The **DARY.A** attribute of the message dialog form points to a text array containing the lines of message text.

The field name returned by **ACCEPT.F** describes which button was pressed. Valid responses returned by **ACCEPT.F** are "OK", "CANCEL", "YES", "NO", "ABORT", "RETRY" and "IGNORE." (**Note:** there is no display entity corresponding to these field names. Do not use **DFIELD.F** on a message dialog box). The following example shows a typical interaction with a message dialog:

```
show MESSAGE.PTR with "retry_cancel.frm"
let TEXT.LINES(1) = "D:\ is not accessible"
let TEXT.LINES(2) = "The device is not ready."
let DARY.A(MESSAGE.PTR) = TEXT.LINES(*)

select case ACCEPT.F(MESSAGE.PTR, 0)
         case "RETRY"    ...
         case "CANCEL"   ...
endselect
```

5.8.3 File Selection Dialog

Toolkits provide standard dialogs for browsing through the directory structure of the file system. These dialogs can now be accessed from within a SIMSCRIPT program using the **FILEBOX.R** routine as:

```
let FILTER = “*.dat”
let TITLE = “Select a data file...”
call FILEBOX.R given FILTER, TITLE yielding PATH.NAME, FILE.NAME
```

The **FILTER** variable can either be a wild card, or a fully or partially qualified file name. The selected file and its path are returned in the **FILE.NAME** and **PATH.NAME** variables. The **TITLE** parameter is the text shown in the title bar of the dialog.

5.8.4 System Font Browser

A predefined dialog box can be brought up programmatically allowing the user to select system font attributes from those available on the server. This is done by calling **FONTBOX.R** as follows:

```
let TITLE = “Select a font”
call FONTBOX.R given  TITLE yielding
```

```
      FAMILY.NAME, POINT.SIZE, ITALIC.DEGREE, BOLDFACE.DEGREE
```

The yielded arguments are identical to those described above for **TEXTSYSFONT.R.** **FONTBOX.R** will not return until a font has been selected, or *cancel* has been pressed. In this case the result of **FAMILY.NAME** will be **""**.

5.8.5 Printing the Contents of a Graphics Window (or Individual Segment)

Microsoft Windows users can programmatically have the contents of either an entire window or an individual segment sent to a printer through the system print dialog box with the following routines:

```
call PRINTWINDOW.R given WINDOW.PTR, USE.DIALOG
   yielding SUCCESS
call PRINT.SEG.R given SEGMENT.ID, USE.DIALOG yielding SUCCESS
```

where **WINDOW.PTR** is the display entity returned from **OPENWINDOW.R**. If the **USE.DIALOG** integer parameter is non-zero, the system print dialog is displayed allowing the user to set print options. **SEGMENT.ID** is either obtained from the **SEGID.A** icon attribute or from **SEGID.V**. The integer **SUCCESS** is non-zero if printing was completed successfully. On UNIX systems, the graphics are written to an EPS PostScript file.

5.9 Menu Bars and Palettes

5.9.1 Menu Bar

A menu bar is composed of several menus arranged in a row on a bar across the screen. Clicking on one causes its menu to be displayed. Clicking on an item inside a menu causes it to be selected. In addition to being returned by **ACCEPT.F,** the index number of the item selected is accessible through the **DDVAL.A** of the particular menu, as in:

```
let ITEM.NUM = DDVAL.A(DFIELD.F(MENU.TXT, FORM.PTR))
```

where the menu name (here a variable) must be all capitals. The indexing begins at 1.

From SIMDRAW, the constructed menu bar can be *cascadeable* (i.e. menus can contain other menus). This hierarchy is preserved in your program with respect to fields accessible by **DFIELD.F**. To access a menu contained within another menu, pass the reference name of the desired sub-menu along with a pointer to the parent menu. **DFIELD.F** will then return a pointer to the sub-menu. Since **DFIELD.F** is recursive, it can be used regardless of how many layers of menus are between the form and the desired field.

You can have check marks displayed programatically next to any menu item. To display the check mark, set the **DDVAL.A** attribute of the menu item field pointer to "1", and then re-display the field. For example:

```
let DDVAL.A(DFIELD.F("MENUITEM_TO_CHECK", MENUBAR.PTR)) = 1
let DDVAL.A(DFIELD.F("MENUITEM_TO_UNCHECK", MENUBAR.PTR)) = 0
display DFIELD.F("MENUITEM_TO_CHECK", MENUBAR.PTR)
display DFIELD.F("MENUITEM_TO_UNCHECK", MENUBAR.PTR)
```

Remember that a menu item can be grayed out or can be deactivated with the **SET.ACTIVATION.R** routine.

```
call SET.ACTIVATION.R(DFIELD.F("MENUITEM_TO_GRAY",MENUBAR.PTR), 0)
```

5.9.2 Palettes

SIMSCRIPT applications can have palettes attached to the sides of the windows. A palette contains rows and columns of square palette buttons. On the face of each palette button is a bitmap icon. This bitmap comes from a **.bmp** file on MS Windows machines and a **.xwd** file on XWindows machines which must be in the same directory as your **graphics.sg2** library file. Palettes are created only by SIMDRAW.

Palettes are used inside your program the same way that menu bars are; by calling **ACCEPT.F** given pointers to both the palettes display entity and the control routine. The **ACCEPT.F** call must be made from within a process and, a control routine must be included to allow the palette to behave asyncronously. Whenever the user clicks on a palette button, the control routine is called given the button's *field name.*

From within SIMDRAW you can define your palette buttons to be one of three varieties: **momentary**, **toggle** or **dragable**. "Momentary" buttons pop back up automatically after being pressed, while "toggle" buttons stay down. Like dialog boxes and menu bars, the buttons in a palette are represented by *fields* of the palette form. The **DDVAL.A** attribute of the "toggle" button field (obtained using **DFIELD.F**) indicates whether the button is currently "down" or "up".

"Dragable" buttons allow the user to click down on a palette button, and then drag its outline to the canvas of the window. When the mouse button is released, the palette's control routine is called. From within the control routine, the drop point can be retrieved through the **LOCATION.A** attributes of **DINPUT.V**. The **DIVAL.A** attribute of **DINPUT.V** will contain the viewing transform number corresponding to that drop location.

Displaying the palette can be done as follows:

```
process PALETTE
...
show FORM.PTR with "mypalette.frm"
let FIELD.ID = ACCEPT.F(FORM.PTR, 'PALETTE.CTRL')
  '' (will suspend this process)
...
```

This is a typical palette control routine:

```
routine PALETTE.CTL given FIELD.NAME, FORM.PTR yielding STATUS

select case FIELD.NAME
  case "INITIALIZE"   '' called once just after ACCEPT.F call

 case "BACKGROUND"  '' called when clicked on window canvas background

  case "MOMENTARY BUTTON"
    write as "A palette button has been pushed...", /
```

```
  case “TOGGLE BUTTON”
    write DDVAL.A(DFIELD.F(FIELD.NAME, FORM.PTR)) as
       “State of toggle button is “, I 3, /

  case "MY_DRAGGABLE_BUTTON"
    let VXFORM.V = DIVAL.A(DINPUT.V)
    display NEW.ICON at with "temp.icn" at
        (LOCACION.X(DINPUT.V), LOCATION.Y(DINPUT.V))
endselect
end
```

5.10 Examples

The first example is a form that might be used for a deposit in the simulation of an automated teller machine. The form has a text box where you enter a name, a value box where you can enter the amount of the deposit, radio buttons that let you choose either the checking or savings account, and a check box that lets you optionally print a record of the transaction.

This example relies on standard field processing to validate that the deposit amount is not negative and to terminate when **Ok** or **Cancel** is selected. It does not need to use a control routine.

```
Main
  Define FORM as a pointer variable     ''Form Pointer
  Define CHECKBOX as a pointer variable    ''Field Pointer

  Show FORM with "atm.frm"              '' Build the form

  If ACCEPT.F(FORM, 0) eq "OK"          '' OK Button was clicked
    Let CHECKBOX = DFIELD.F("STATEMENT", FORM)
    If DDVAL.A(CHECKBOX) ne 0
      Write DDVAL.A(DFIELD.F("AMOUNT", FORM)) as d (6, 2)
      If DDVAL.A(DFIELD.F("SAVINGS", FORM)) ne 0
        Write as " Deposited to savings account "
      Else
        Write as " Deposited to checking account "
      Endif
      Write DTVAL.A(DFIELD.F("ACCOUNT NAME", FORM)) as t *, /
    Endif
  Endif

End
```

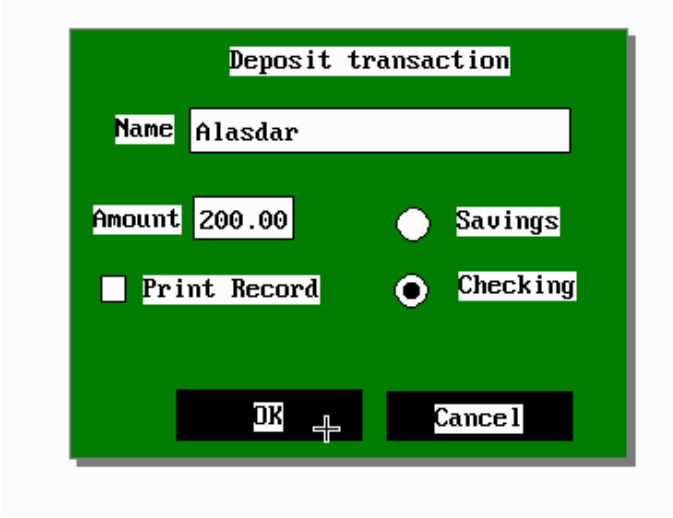


Figure 5-1. Form for the ATM Example

Following convention, only the two button fields, **OK** and **CANCEL**, can terminate the interaction. These button attributes are set in the editor. Action is taken only if the program detects that **OK** was clicked. The fields may be clicked and edited in any order until the user is satisfied with the settings. On completion, the checkbox field is queried to determine whether a transaction record is to be printed. The prototype form used radio buttons to select between two alternatives: savings or checking account. Since only `SAVINGS` is queried, another check box could be substituted without changing the program code.

The next example, List1, creates a form that lets you pick a name from a list. The name will be highlighted when picked. Scroll bars let you scroll through the names to find the one you want. Again, **Ok** or **Cancel** will terminate the selection.

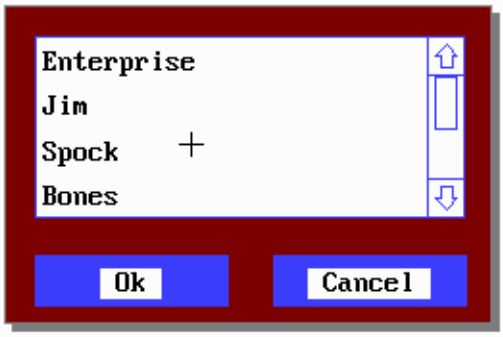


Figure 5-2. Form for List1 Example

```
Main
  Define FORM as a pointer variable     ''Form Pointer
  Define FIELD as a pointer variable    ''Field Pointer
  Define FIELD.ID as a text variable    ''Field ID
  Define NUMBER.OF.NAMES, I as integer VARIABLES
  Define NAMES as a 1-dimensional text array

  Open 1 for input, name is "names.dat"
  Read NUMBER.OF.NAMES using 1
  Reserve NAMES(*) as NUMBER.OF.NAMES   '' Create an array of names
  For I = 1 to NUMBER.OF.NAMES
    Read NAMES(I) using 1

  Show FORM with "list1.frm"            '' Build the form

  Let FIELD = DFIELD.F("MYLIST",FORM)   '' Identify List Box field
  Let DARY.A(FIELD) = NAMES(*)          '' Initialize with names

  Let FIELD.ID = ACCEPT.F(FORM, 0)      '' Request user input

  If FIELD.ID ne "CANCEL"               '' OK Button was clicked
    Let I = DDVAL.A(FIELD)
    If I ne 0
      Write NAMES(I) as "The selected name is ", t *, /
    Endif
  Else                                  '' CANCEL was clicked
    Write as "No selection was made", /
  Endif

End
```

Note the check for a zero value of **DDVAL.A**. The user could have clicked on **OK** without making any selection in the list box! Be aware that the list box field is given a copy of the array pointer, **NAMES(*)**. If a program releases such an array, the **DARY.A** attribute of the list box should be reset to zero, or reinitialized with a new array. Correspondingly, if the list box is destroyed by any means, any array pointed to by **DARY.A** will be released. Any subsequent reference in the program would be invalid.

The next example, List2, is similar to List1, but uses a control routine. By checking that a valid selection has been made when **OK** is clicked, it removes the need to check for a non-zero selection from the MAIN program. This example also shows how the control routine can provide pre-processing of the form data by acting on the INITIALIZE call. No automatic TERMINATE call is required. Any appropriate action can be provided when processing the terminating fields.

```
Main
  Define FORM as a pointer variable     ''Form Pointer
  Define FIELD as a pointer variable    ''Field Pointer
  Define FIELD.ID as a text variable    ''Field ID
  Define NUMBER.OF.NAMES, I as integer VARIABLES
  Define NAMES as a 1-dimensional text array

  Open 1 for input, name is "names.dat", noerror
  Use 1 for input
  If ROPENERR.V eq 0
    Read NUMBER.OF.NAMES
    Reserve NAMES(*) as NUMBER.OF.NAMES   '' Create an array of names
    For I = 1 to NUMBER.OF.NAMES
```

```
      Read NAMES(I)
    Endif
    Close 1

    Show FORM with "list1.frm"            '' Build the form

    Let FIELD = DFIELD.F("MYLIST",FORM)   '' Identify List Box field
    Let DARY.A(FIELD) = NAMES(*)          '' Initialize with names

    Let FIELD.ID = ACCEPT.F(FORM, 'MYLIST.CTRL')  '' Request input

    If FIELD.ID ne "CANCEL"               '' OK Button was clicked
      Let I = DDVAL.A(FIELD)
      Write NAMES(I) as "The selected name is ", t *, /
    Endif
End
Routine MYLIST.CTRL Given FIELD.ID and FORM Yielding STATUS

    Define FIELD.ID as a text variable
    Define FORM as a pointer variable
    Define STATUS as an integer variable  ''Accept/Reject Data
    Define .BEEP to mean 7  ''ASCII <bel> character value

    Select case FIELD.ID
     Case "INITIALIZE"        '' Default to select 1st entry
      If DARY.A(DFIELD.F("MYLIST",FORM)) ne 0
        Let DDVAL.A(DFIELD.F("MYLIST",FORM)) = 1
      Endif
     Case "OK"
      If DDVAL.A(DFIELD.F("MYLIST",FORM)) eq 0
        ''No item is selected; beep and do not terminate
        Write .BEEP as a 1, + using 6
        Let STATUS = -1
      Endif

     Default
    Endselect

End ''MYLIST.CTRL
```

6. Creating Animated Graphics

Both static and dynamic (moving) icons can be created using SIMGRAPHICS II. These icons are screen images of SIMSCRIPT II.5 program entities, and changes in their position or appearance correspond to changes in the system being simulated. For example, parts can accumulate in a manufacturing queue, and partially-completed products can move from station to station in a production line model; ships can arrive at a dock or wait at sea for a tugboat to escort them; and so on.

Further, static or dynamic icons can change color or shape to indicate a change in status: an idle machine could be shown in blue, a busy machine in green, and a broken machine in red. Similarly, icons can change shape to indicate status: the icon for a busy conveyor could include its cargo, for example. These effects can be combined to produce meaningful backgrounds and displays that bring the results of simulation to life.

There are two ways to create icons, by drawing them on the screen with a mouse using the Icon environment within SIMDRAW, or by specifying the coordinates as a set of points which are then connected, filled and colored by library routines such as **`FILLAREA.R`** and **`LINECOLOR.R`** (Appendix A). SIMDRAW is the preferred means of icon manipulation and is described below. The connect-the-dots method is described in Chapter 9.

The basic steps in adding animation to a SIMSCRIPT II.5 simulation are:

1. Decide what to animate and define those entities as `GRAPHIC` in the program preamble;
2. Define visual icons to represent the program entities on the screen, using SIMDRAW;
3. Translate the model's world into a screen world by defining a coordinate system; (Define this same coordinate system in SIMDRAW)
4. Add `DISPLAY` and animation control statements to your code.

Each step is described, in turn, below.

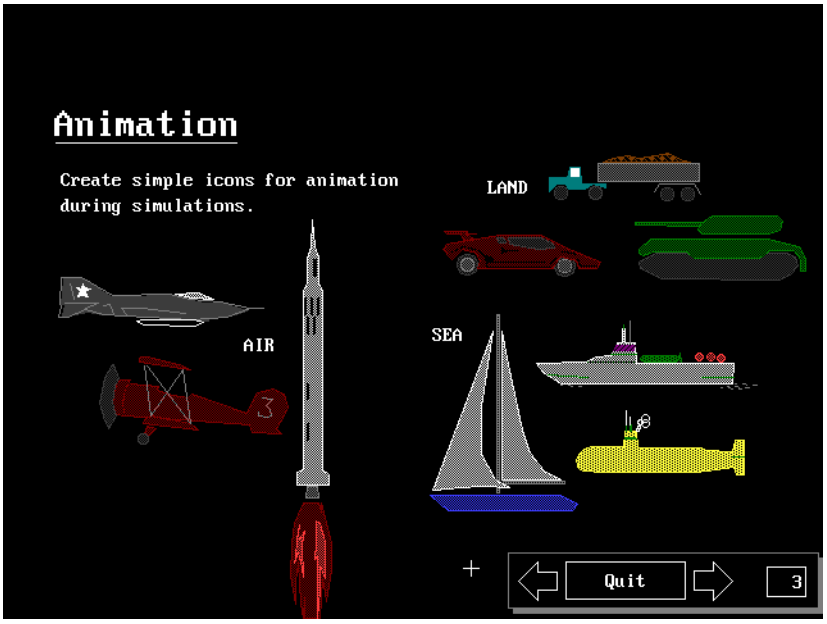


Figure 6-1. Animated Icons

6.1 Graphic Entity Declaration

The principal elementary objects in any SIMSCRIPT simulation program are processes and temporary entities. SIMGRAPHICS II provides an easy way to associate graphical images with the entities in a model.

SIMGRAPHICS II supports both `GRAPHIC` entities and `DYNAMIC GRAPHIC` entities. `DYNAMIC GRAPHIC` entities can move across the screen. `GRAPHIC` entities are motionless. Any temporary entity, including processes, may be declared to be `GRAPHIC` by adding the following statement to the program preamble:

```
[DYNAMIC] GRAPHIC ENTITIES INCLUDE name1 [, name2 ] ...
```

This statement may be placed anywhere after the entity definition in the preamble. Declaring an entity as `GRAPHIC` or `DYNAMIC GRAPHIC` automatically provides additional attributes that are used by SIMGRAPHICS II to maintain the graphical image.

6.2 Coordinate Systems

SIMGRAPHICS II has no inherent measurement units. Many applications have their own geometry and are easier to design and understand if all measurements are given in the relevant units (miles or meters, for example). Other applications may use the Y-axis to represent some measured value, such as queue length, and the X-axis to represent some other unit, such as time.

Spaces in SIMGRAPHICS II are thus defined in a model-oriented manner, in real-world coordinate units. These are transformed into coordinates suitable for graphics display through the viewing transformations (figure 6-2) described below.

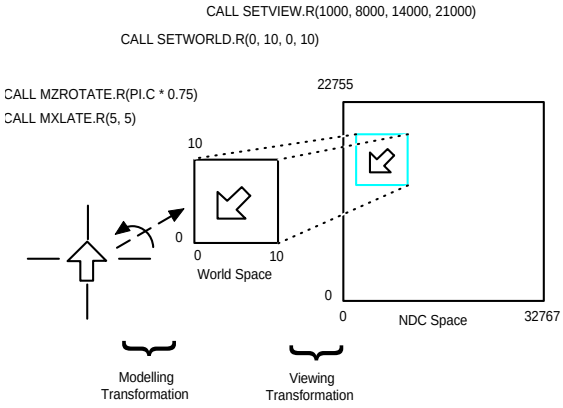


Figure 6-2. Coordinate Transformations

6.2.1 Normalized Device Coordinates

Because the resolution of display screens vary, the coordinate points of shapes to be drawn are specified in Normalized Device Coordinates (NDC units), which represent a mapping of an arbitrary space onto available pixels. NDC units range from 0 to 32767 along both the X- and Y-axes. Scaling in both directions is assumed to be the same, so that squares and circles can be easily specified. In the NDC system of notation, the origin point (0,0) is located in the lower-left corner of the display surface. The lower-right corner has the coordinates (32767,0), and the upper-left corner is (0,32767).

The default space, or viewport, provided by SIMGRAPHICS II is a one-to-one mapping of real-world coordinates into NDC units. Objects that overlap the edge of the screen are partially displayed (clipped). The units thus conform to the aspect ratios used by many display devices.

6.2.2 Setting a Viewing Transformation

The NDC system provides a default space in which a display may be designed. However, the world space in which the simulated system operates differs from model to model. In one case, the action may take place on a factory floor; in others, an entire continent or the bus lines on a printed circuit board may be involved. All these displays are presented on a standard screen, so some variable scaling is obviously desired.

It is convenient for the program to specify a region in its world space which is to be portrayed on the display as a window onto world space. The graphics system can then calculate the scaling required and apply it to each graphic primitive, in order to represent the region of world space on the display. By altering the parameters of this transformation, the

display can appear to zoom in or out, or to pan up, down, or sideways, displaying different regions of world-space.

Users may create Cartesian Coordinate systems scaled to their own applications by defining viewing transformations. SIMGRAPHICS II permits up to 15 user-defined mappings, indicated by setting the variable **VXFORM.V** to a value between 1 and 15. (The default transformation, **VXFORM.V = 0**, represents the entire NDC space described above and cannot be changed.)

Thus, for example, a representation of some physical layout may coexist on the display with a menu of choices. Clearly the transformation appropriate for objects in the layout is different from that applied to the wording on the menu. Further, the shape of an icon can be specified in NDC units while its position is given in terms of a transformation. This makes it possible to zoom in or out on the display without the icon changing size.

Each **VXFORM.V** transformation is defined or redefined with calls on the system routines **SETWORLD.R** and **SETVIEW.R** (described below). The program must set **VXFORM.V** before calling these routines. Multiple transformations may operate in different units, and they may overlap. For instance, a factory floor, measured in meters, can map into a viewport on part of the display screen, while a menu is displayed on another part.

A particularly convenient transform is one which is 80 wide by 24 high with its origin at the top left. This transformation can be used to map text to the screen in a way which is analogous to standard text screens. Thus, the following statement prepares to write at column 65 of row 18:

```
LET VXFORM.V = 1
CALL SETWORLD (0, 79, -23, 0)
CALL MXLATE.R (65, 18)
```

6.2.3 Defining The World: SETWORLD.R

Routine **SETWORLD.R** establishes the coordinates of a space, given in real-world units. These coordinates are converted into device coordinates (NDC units) through parameters given to the routine **SETVIEW.R**. **SETWORLD.R** defines a rectangle in real world space that is to be mapped into the NDC space. Areas and points outside the area are clipped. This routine is called with the statement:

```
CALL SETWORLD.R (w.xlo, w.xhi, w.ylo, w.yhi)
```

where `w.xlo` and `w.xhi` are real numbers, in real world coordinates, defining the limits of the x-axis, and `w.ylo` and `w.yhi` are real numbers, in real world coordinates, defining the limits of the y-axis; `w.xlo` ≤ `w.xhi` and `w.ylo` ≤ `w.yhi`.

The argument values establish the area to be mapped onto the display surface. For example, in the Gold Mine problem (Chapter 7), the display is centered around the top of a mine shaft and extends 50 feet to the left of the shaft and 200 feet to the right, 220 feet underground and 20 feet above ground. Thus, the statements:

```
LET VXFORM.V = 1
CALL SETWORLD.R(-50.0,200.0,-200.0,20.0)
```

would appear in the main routine.

6.2.4 Defining a Viewport: `Routine SETVIEW.R`

Routine **`SETVIEW.R`** defines a viewport rectangle on the display surface, using NDC units. The area defined by **`SETWORLD.R`** is uniformly mapped into this area. Calls take the form:

```
CALL SETVIEW.R (v.xlo, v.xhi, v.ylo, v.yhi)
```

where `v.xlo` and `v.xhi` are integers in Normalized Device Coordinate units, $0 \le$ `v.xlo` $<$ `v.xhi` $\le$ `32767`; and `v.ylo` and `v.yhi` are integers in Normalized Device Coordinate units, $0 \le$ `v.ylo` $<$ `v.yhi` $\le$ 32767. This routine defaults to full window display. It needs be called directly only if a smaller display is desired.

`SETVIEW.R` and **`SETWORLD.R`** operate with rectangular areas. In many applications, the X- and Y-axes are scaled in the same units. To avoid distorting square objects, make sure that the parameters given to **`SETWORLD.R`** and **`SETVIEW.R`** map squares in real world coordinates into squares in Normalized Device Coordinates. The squareness requirement can be mathematically stated as:

```
abs.f((w.xhi - w.xlo)/(v.xhi - v.xlo))
=abs.f((w.yhi - w.ylo)/(v.yhi - v.ylo))
```

This relationship uses the fact that NDC units in SIMGRAPHICS II preserve the aspect ratio. An easier way is just to use a square worldview with a square viewport on part of the screen. This also frees part of the screen for other uses.

Although the two calls interact, **`SETWORLD.R`** and **`SETVIEW.R`** may be called in any order. Subsequent calls on either or both routines can be used to partially or totally redefine a viewing transform. This property can be used very effectively to perform a zoom operation.

Note that whenever **`SETWORLD.R`** and **`SETVIEW.R`** are called, all objects drawn under the current viewing transformation are automatically redisplayed. If you wish to change both the world coordinate system and the viewport, then bracket the calls to **`SETWORLD.R`** and **`SETVIEW.R`** by calls to **`GDEFERRAL.R`** as follows:

```
call GDEFERRAL.R(1)
call SETWORLD.R(w.xlo, w.xhi, w.ylo, w.yhi)
call SETVIEW.R(v.xlo, v.xhi, v.ylo, v.yhi)
call GDEFERRAL.R(0)
```

6.2.5 Modelling Transformations

In addition to the viewing transformation described above, the ambitious programmer may use a modelling transformation to specify or change an object's location or to rotate an object around its origin (figure 6-3). The SIMGRAPHICS II system uses modeling transformations to produce animated effects. Modeling transformations also make it easy to display copies of static objects in various positions in world space by specifying a master pattern for the image with its “virtual position” at the origin. The modeling transformation then produces a copy of the object in the desired position and orientation.

Every graphic entity is automatically provided with attributes to control its modeling transformation. These attributes are:

`ORIENTATION.A (entity)`	Orientation (in radians), counterclockwise from 3 o'clock.
`LOCATION.A (entity)`	Location of the object with respect to its origin.

Values may be assigned directly to **`ORIENTATION.A`**. **`LOCATION.A`** must be set using the value produced by the system function **`LOCATION.F (xpos, ypos)`** as follows:

```
LET LOCATION.A (entity) = LOCATION.F (xpos, ypos)
```

where *xpos* and *ypos* are real world coordinates. Changing the location causes the image to be redrawn automatically, so change orientation before changing location.

You can specify the origin of your icon with respect to the points which define it from within SIMDRAW. When in the **Image Editor** use the **Edit/Image...** option. Click on the **Select...** button inside the **Image Attributes** dialog, and then point to where you want the origin to be. The object will be rotated about this point if the **`ORIENTATION.A`** attribute is assigned in your program.

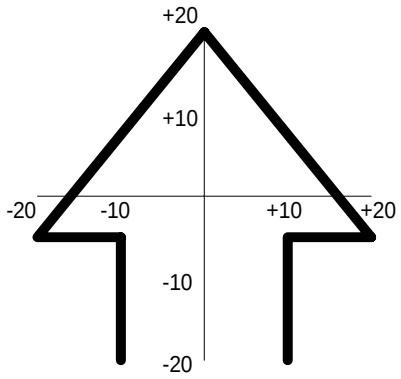


Figure 6-3. Object Origin

The modelling transformation is supported by the library routines:

`MXRESET.R (entity)`	Set modeling transform from an entity.
`MZROTATE.R (radians)`	Rotate counterclockwise around origin.
`MXLATE.R (xval, yval)`	Translate (move) in X and Y.

(System attributes, functions, and routines are completely described in Appendix A). These are called as a standard part of the `DISPLAY` routine . Their explicit use may be necessary in certain advanced situations. Rotation is performed before translation. The effects of successive calls on both **`MZROTATE.R`** and **`MXLATE.R`** are cumulative.

The modelling transformation is applied before the viewing transformation specified by **`VXFORM.V`**. The modelling transformation is logically separate from the viewing transformation.

6.3 Animating Dynamic Graphic Entities

Dynamic graphic entities follow the same rules as static graphic entities, but have additional display features which manipulate their location, orientation, and velocity. The dynamic nature of such entities is controlled by giving them a velocity. As simulation time progresses, the location is automatically updated as determined by the velocity, causing the entity to be redrawn on the display surface.

Control of animation is provided by the system function **VELOCITY.A (*entity*)**, a left function that sets velocity. Except for the special value of 0, which stops linear motion, the value of **VELOCITY.A** must be set to a value produced by the system function **VELOCITY.F (*speed, theta*)**, where ***speed*** is a real value in Real World Coordinate Units per Simulated Time Units, and ***theta*** is the direction of motion in radians. For example, in the Gold Mine problem (Chapter 7), the mine elevator travels straight down at a speed of 33.33 feet/minute. Thus, the statement:

```
LET VELOCITY.A(lift) = VELOCITY.F(100/3,-PI.C/2)
```

appears in the code.

Note that motion continues until an entity is explicitly stopped with the statement:

```
LET VELOCITY.A(entity) = 0
```

Assigning a value of **VELOCITY.F(0., 0.)** to **VELOCITY.A** makes an object mark time. Its image is refreshed as simulated time passes, but the position is not updated. This can be useful for objects whose animation consists of something other than simple linear motion.

VELOCITY.A changes the value of **LOCATION.A** as simulated time changes. An object's starting position is obtained by setting **LOCATION.A**.

6.4 Displaying Icons

In order to display an object, execute the DISPLAY statement:

```
DISPLAY name1 [WITH "iconname"] [AT (posx, posy)]
```

where name1 is a graphics entity pointer. Or use the SHOW statement:

```
SHOW name1 [WITH "iconname"] [AT (posx, posy)]
```

These statements may be placed anywhere in the executable flow of program control.

If the DISPLAY statement is executed, any previous image of the object is removed from the display surface, and the new image appears at the screen location set by **(posx, posy)** above or a **LOCATION.F** statement.

The SHOW statement by itself does not make an icon visible, but rather associates the object with an icon. To make the icon visible after a SHOW statement, either the explicit statement:

```
DISPLAY name
```

or assignment to **LOCATION.A** must be executed.

6.5 An Example

The animated graphical features will now be illustrated in the context of a complete simulation example. The simulation aspects of this model have been simplified to focus attention on the graphical interactions.

6.5.1 Preamble

This preamble defines one process (called **shape**) and declares it to be a `DYNAMIC GRAPHIC` entity. The icon for **shape** was prepared using the **Icon Editor**, and can be any shape whatsoever. For this example, it was drawn as an arrow.

```
Preamble   ''Case Study "NEWSHAPE"

''   A simple dynamic graphics output using SIMGRAPHICS II.
''   It draws a shape and moves it around the screen.

     Normally mode is undefined

     Processes include SHAPE

     Dynamic graphic entities include SHAPE

End ''Preamble
```

Figure 6-4. Output of the Shape Routine

6.5.2 Main Program

Prior to starting the animated simulation, there are some initial conditions which must be established. Lines 3 and 4 set up the viewport and dimension the world to be seen in that viewport. Line 7 determines how simulated time should be scaled to real time (see paragraph 6.7). Finally, to begin the simulation, one **shape** is activated and assigned its icon.

```
Main
''   Set up the world view and view port
     Let vxform.v = 1          '' View port number
     Call setworld.r(0.0, 2000.0,0.0, 2000.0)

''   1 second of real time per second of simulated time
     Let TIMESCALE.V = 100

''   Create a shape and specify the icon for it
     Activate a SHAPE now
     Show SHAPE with "shape"

     Start simulation
End ''Main
```

6.5.3 Process Shape

The **shape** process describes the life of the object. Normally, this module would contain much more complex logic to describe its interaction with other objects, use of resources, etc. In this example, it merely experiences a series of time delays interspersed with changes in direction of motion. The setting of the **VELOCITY.A** attribute causes the animation to appear.

This program is complete and will execute as is. The only thing not listed here is the output from SIMDRAW. The program is included on the distribution disks for SIMSCRIPT as NEWSHAPE and may be run from there. We recommend that a new user of SIMGRAPHICS II run this model as is and then modify it to do other things. (For example, you might draw a different shape or change the path the object follows.)

A version of this same model which does not use SIMDRAW output is included in Chapter 9. It is instructive to compare the two models to see how much code is eliminated through use of the editor.

```
Process SHAPE
     Define I as an integer variable

''   Set up the parameters for controlling motion
     Let velocity.a(SHAPE) = velocity.f(200.0, pi.c/4)
     Let location.a(SHAPE) = location.f(0.0, 0.0)

''   Make the first move
     Wait 10 units

''   Change the direction of motion to straight down
     Let velocity.a(SHAPE) = velocity.f(200.0, -pi.c / 2)
     Wait 5 units

''   Change the direction of motion again
     Let velocity.a(SHAPE) = velocity.f(200.0, 0.8*pi.c)
''   Make the shape rotate
     For I = 1 to  60 do
          Add PI.C / 60 to orientation.a(SHAPE)''make it tumble
          Wait 0.1 units
     Loop
```

```
''   Stop the movement and pause to admire the results
     Let velocity.a(SHAPE) = 0
     Wait 5.0 units

End '' process SHAPE
```

6.6 Destroying and Erasing Icons

The image of a graphic entity is placed on the display surface when the first DISPLAY statement is executed. Subsequent execution of a DESTROY statement causes the associated image to be removed from the display surface. All extra memory utilized for graphics data is reclaimed when the DESTROY statement is executed. The ERASE statement merely removes an image from the screen.

6.7 Synchronizing Simulation Time and Real Time

Synchronization between real time and simulation time in SIMGRAPHICS II is facilitated with the real global variable **TIMESCALE.V**. The value of **TIMESCALE.V** establishes a scaling between real-time (in units of 1/100 second) and simulation time. For example, the statement:

```
LET TIMESCALE.V = 100
```

establishes a one-to-one mapping of simulation time units and real elapsed seconds—if the computer can work that fast.

Examples are:

TIMESCALE.V	Simulated Time Units	Real Time
1	1	0.01 second
1	100	1 second
100	1	1 second
6000	1	60 seconds

SIMGRAPHICS II updates the display of all dynamic graphic entities whenever the simulation time clock is updated. If simulation time is passing faster than wall clock time, the displays are updated at more frequent intervals. If the desired rate is faster than processing speed allows, SIMGRAPHICS II will be unable to catch up.

Decreasing the value of **TIMESCALE.V** has the effect of making the simulation run faster, in less elapsed time, provided there is enough computer power to do both the computational simulation and the animated graphics. When there is not enough computer power, additional elapsed real time will be taken.

The timing process of SIMGRAPHICS II also supports user intervention in the process, through the global variable **TIMESYNC.V**. **TIMESYNC.V** is a subprogram variable which, when non-zero, points to a user-supplied time synchronization exit routine. This routine is called from the event scanning mechanism offering the simulation time of the most imminent event.

The TIMESYNC routine has the option of consulting the computer's real time clock and substituting a lower value to be used to update simulation time. This forces re-scanning of the event list until the TIMESYNC routine is prepared to accept the tendered value of **TIME.V**. Re-scanning the event list allows the possibility that the future event may change either as a result of some action in the TIMESYNC routine, or completion of an asynchronous read request. During this activity, of course, the routine can choose to provide a continually updated display of simulation time, in any appropriate format.

The user's exit routine is called whenever the simulated clock is to be updated, but before any animation is performed. This routine may perform several functions, including providing a graphical display of simulated time, fine-detail time synchronization or adjustment, scheduling or canceling events or processes, or collision avoidance computations. The first is the most common use.

The **TIMESYNC.V** exit can serve as a convenient means for adjusting the flow of real time and simulated time, to improve Real Time control. When lockstep operation is desired, **TIMESYNC.V** can reschedule heavy computation or adjust the detail of the simulation as appropriate.

The value of **TIMESCALE.V** could also be adjusted to smoothly catch up after heavy computation. By using the library routine **SYSTIME.R**, the program can compare the current simulation time to the simulation time which should have been advanced to at the current real time. The program can then adjust **TIMESCALE.V**, accordingly.

7. Example Programs

This chapter describes the programs included in the SimLab distribution kit. They are actual SIMSCRIPT II.5 simulations in which graphic effects of various kinds have been added.

7.1 The Gold Mine Program

The Gold Mine model simulates the operation of a two level mine. Ore moves from the shafts to the surface in a single elevator that serves both levels. Loads from the lower level have priority over loads from the upper level.

Only a few routines from the model are presented here. The rest of the application is included in the distribution kit. It includes animated graphics on a static background, along with a variety of presentation graphics. Note that the menu bar in the example is asynchronous—the simulation parameters may be altered on the fly. All graphics were created using SIMDRAW. Figure 7-1 shows the Gold Mine simulation running.

In this model, the lift is the only dynamic graphic entity. The loads are graphic entities but their motion is controlled by the lift. Consequently, the lift must have a user-written display routine (Chapter 9) to control the movement of the loads onto and off of the lift.

The asynchronous menu bar allows the user to alter the presentation graphics displayed as the model runs (a clock, and two plots of the queues at the two levels). It also lets the user terminate the simulation or alter the rate of ore arrival and the lift capacity. The queue length on each level is displayed as either a trace plot or a dial or level meter. Of course, these presentation graphics could be further modified by use of the **Graph Editor**.

The clock is updated through the **`TIMESYNC.V`** mechanism, and time is allowed to take discrete jumps since the events are occurring rapidly enough to appear relatively smooth.

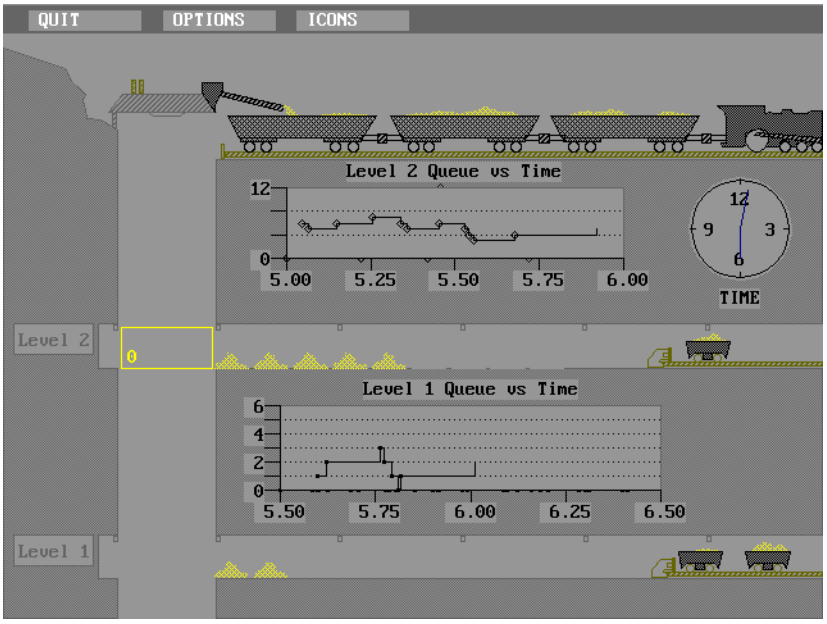


Figure 7-1. The Gold Mine

The motion of the lift is controlled entirely from within the lift process. Since the direction is either up or down, no computations of coordinate geometry are required in this model.

Listed below are the process, control routine, and form dump of the menu bar. They are included here because of the asynchronous nature of the input. The complete program is included in all distributions.

7.1.1 Menu Bar Process

```
Process MENUPROC

  Define FIELD.ID as text variable
  Define MENU as pointer variable
  Define MENUCURSOR as pointer variable


  Show MENU with "goldmenu.frm"
  Let FIELD.ID = ACCEPT.F(MENU,'MENUBAR.CTRL')
  Let FIELD.ID = FIELD.ID

  Activate a FINAL.REPORT now

End ''MENUPROC
```

7.1.2 Form Control Routine

```
Routine MENUBAR.CTRL Given FIELD.ID and FORM Yielding STATUS

  Define DIALOG as a pointer variable
```

```
Define FIELD.ID as text variable
Define FORM as pointer variable
Define ICONS as a 1-dim text array
Define STATUS as integer variable


''If the menubar was selected, then find out which button on the bar
''or a pulldown was pressed.
If FIELD.ID eq "MENU"
  Let FIELD.ID = DTVAL.A(DFIELD.F("MENU",FORM))
Endif

Select case FIELD.ID
 Case "INITIALIZE"

 Case "QUIT"
  Let STATUS = 1

 Case "OPTIONS"

 Case "ICONS"

 Case "LIFT CAP"
   Show DIALOG with "liftcap.frm"
   Let DDVAL.A(DFIELD.F("CAPACITY",DIALOG)) = LIFT.CAPACITY

   Let FIELD.ID = ACCEPT.F(DIALOG,0)

   If FIELD.ID ne "CANCEL"
     Let LIFT.CAPACITY = DDVAL.A(DFIELD.F("CAPACITY",DIALOG))
   Endif

 Case "ARRIVAL 1"
   Show DIALOG with "arrive.frm"
   Let DDVAL.A(DFIELD.F("ARRIVE",DIALOG)) = MEAN.RATE(1)

   Let FIELD.ID = ACCEPT.F(DIALOG,0)

   If FIELD.ID ne "CANCEL"
     Let MEAN.RATE(1) = DDVAL.A(DFIELD.F("ARRIVE",DIALOG))
   Endif

 Case "ARRIVAL 2"
   Show DIALOG with "arrive.frm"
   Let DDVAL.A(DFIELD.F("ARRIVE",DIALOG)) = MEAN.RATE(2)

   Let FIELD.ID = ACCEPT.F(DIALOG,0)

   If FIELD.ID ne "CANCEL"
     Let MEAN.RATE(2) = DDVAL.A(DFIELD.F("ARRIVE",DIALOG))
   Endif

 Case "TIME"
   Show DIALOG with "time.frm"
   Let DDVAL.A(DFIELD.F("SCALE",DIALOG)) = SCALER
   let DDVAL.A(DFIELD.F("LENGTH",DIALOG)) = STOP.TIME

   Let FIELD.ID = ACCEPT.F(DIALOG,0)

   If FIELD.ID ne "CANCEL"
     Let SCALER = DDVAL.A(DFIELD.F("SCALE",DIALOG))
```

```
                                    ''clock ticks (1/100 sec) / unit

        Let TIMESCALE.V = SCALER*100/60.0
        let STOP.TIME = DDVAL.A(DFIELD.F("LENGTH",DIALOG))  ''hours
      Endif

    Case "QUEUE 1"
      Show DIALOG with "icon.frm"
      Reserve ICONS as 3
      Let ICONS(1) = "trace1.grf"
      Let ICONS(2) = "level1.grf"
      Let ICONS(3) = "dial1.grf"
      Let DARY.A(DFIELD.F("PICK",DIALOG)) = ICONS(*)

      Let FIELD.ID = ACCEPT.F(DIALOG,0)

      If FIELD.ID ne "CANCEL" and DDVAL.A(DFIELD.F("PICK",DIALOG)) ne 0
        Let ICONS(*) = DARY.A(DFIELD.F("PICK",DIALOG))
        Let QUEUE.ICON(1) = ICONS(DDVAL.A(DFIELD.F("PICK",DIALOG)))

        ''Load the new graph
        Display N.LOAD.QUEUE(1) with QUEUE.ICON(1)
      Endif

    Case "QUEUE 2"
      Show DIALOG with "icon.frm"
      Reserve ICONS as 3
      Let ICONS(1) = "trace2.grf"
      Let ICONS(2) = "level2.grf"
      Let ICONS(3) = "dial2.grf"
      Let DARY.A(DFIELD.F("PICK",DIALOG)) = ICONS(*)

      Let FIELD.ID = ACCEPT.F(DIALOG,0)

      If FIELD.ID ne "CANCEL" and DDVAL.A(DFIELD.F("PICK",DIALOG)) ne 0
        Let ICONS(*) = DARY.A(DFIELD.F("PICK",DIALOG))
        Let QUEUE.ICON(2) = ICONS(DDVAL.A(DFIELD.F("PICK",DIALOG)))

        ''Load the new graph
        Display N.LOAD.QUEUE(2) with QUEUE.ICON(2)
      Endif

    Default
   Endselect

End ''MENUBAR.CTRL
```

7.2 The DYNHIST Model

DYNHIST illustrates the use of dynamic histograms. Samples from a uniform distribution and samples from a normal distribution are displayed as histograms. In addition, a level meter is used to display the number of samples taken. This type of display can be used alone or can be included in the margins of an animated display. A sample of its output is shown in figure 7-2.

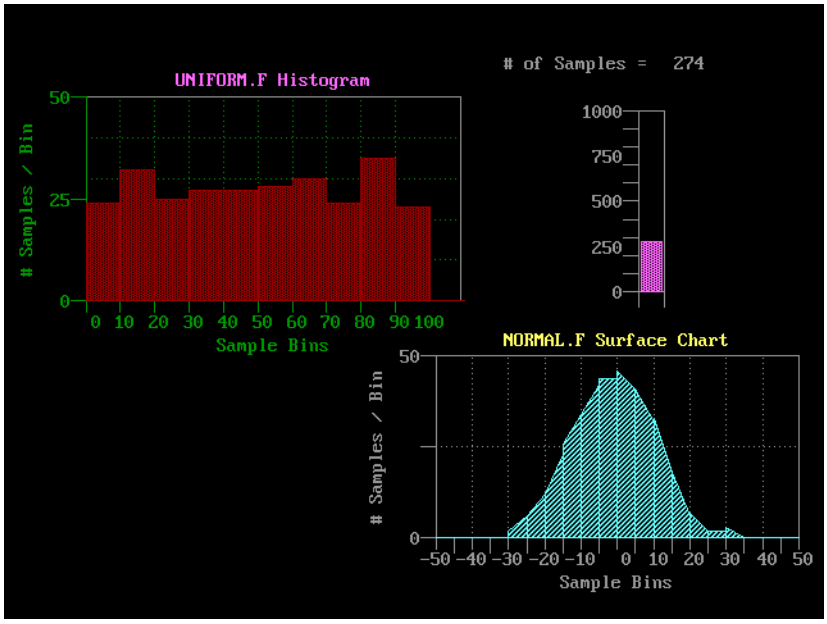


Figure 7-2. Output of the DYNHIST Model

7.3 The Port Model

The African Port has been modelled in many languages in many simulation textbooks. This is the SIMSCRIPT II.5 version with animation. Several important animation concepts are illustrated in it. In particular, segmentation is used to prioritize passing ships. Color is used to differentiate different classes of ships and to highlight ships which need attention. Output is shown in figure 7-3.

The animation reveals a logic flow error in this model. On occasion, the tug makes an unnecessary dead-heading trip when it could just as well have served a ship in its present location. The reader may find it an instructive exercise to find the error.

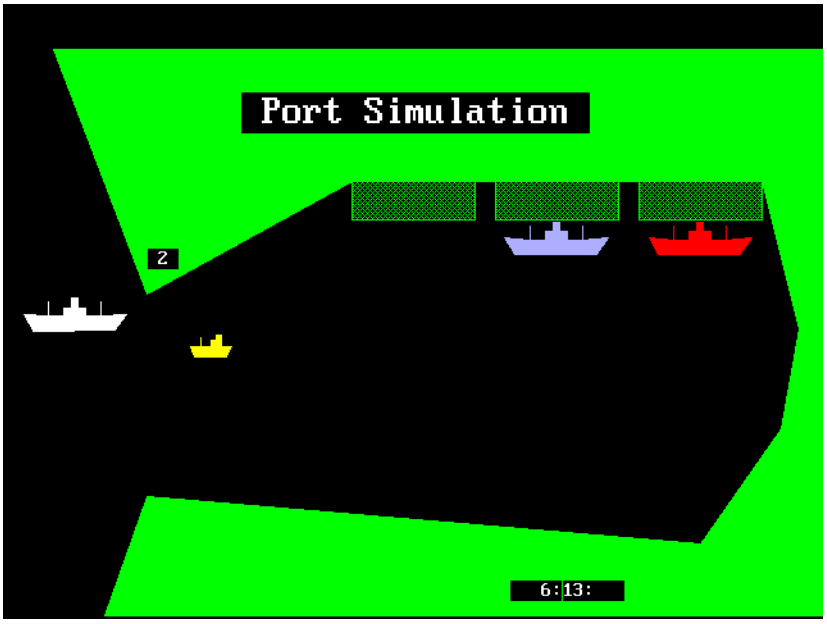


Figure 7-3. The Port Model

7.4 The CALSHIP Model

This model is an expansion of the PORT model above. It illustrates the relative ease of extending a simple model to include more of the entire system. In this case, the simple port model is duplicated to represent two ports (Valdez, Alaska and San Pedro, California) between which tankers travel. Each port is shown in a separate window. The entire coastline is shown in another window. This window also shows tanker traffic from Indonesia coming to San Pedro. Output is illustrated in figure 7-4.

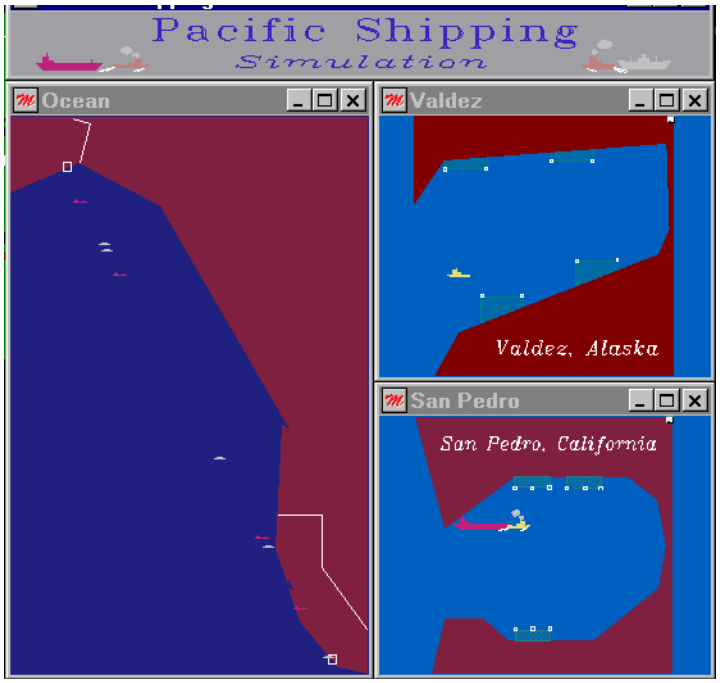


Figure 7-4. The CALSHIP Model

7.5 The Spring Model

This model introduces the use of animation in conjunction with the continuous simulation constructs of SIMSCRIPT II.5. Plots of displacement, velocity, and acceleration (figure 7-5) are produced simultaneously.

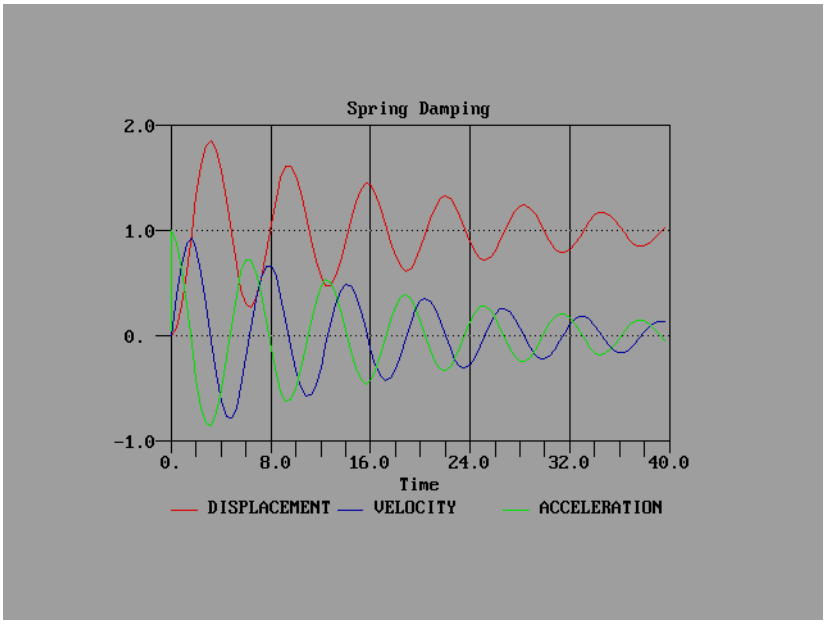


Figure 7-5. The Spring Model

7.6 The Pilot Ejection Model

EJECT is derived from a classic example of continuous simulation. A pilot ejects from a plane at low altitude. Dials display the pilot's speed in the X and Y directions and the time since ejection. Animation is used to display the position of the plane, the pilot, and the deployment of his parachute. The continuous simulation constructs of SIMSCRIPT II.5 are used to implement the set of differential equations describing the motion of each object. Output is shown in figure 7-6.

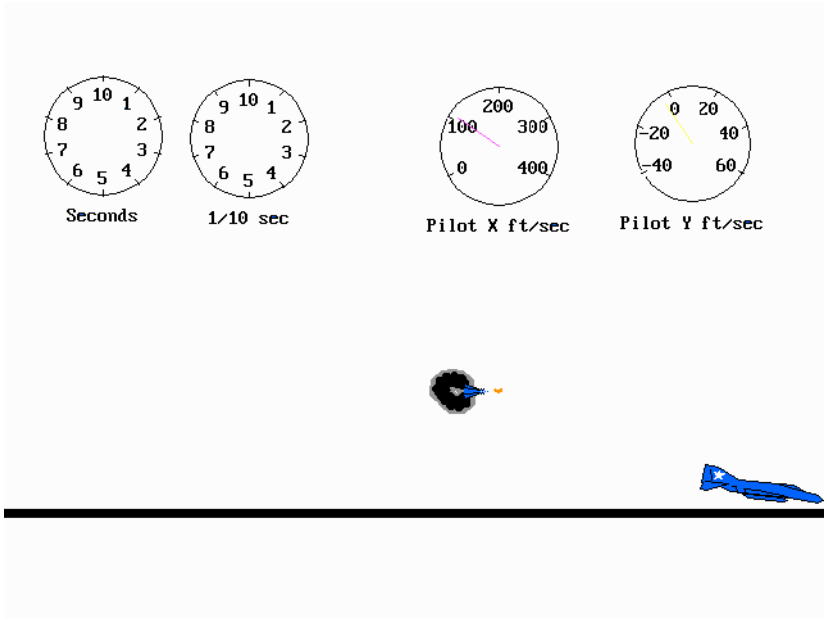


Figure 7-6. The EJECT Model

8. Managing Multiple Windows

8.1 Multiple Window Support

SIMGRAPHICS II can now give the SIMSCRIPT programmer multiple windows with various sizes, positions, titles, and mapping styles. Each window can optionally have a horizontal and vertical scroll bar, and a multi-pane status bar. In addition, messages are passed from the window manager to your program whenever the user manipulates a component of the window, (i.e. resizing, closing, moving the thumb on a scroll bar, etc.).

A window is created with the **OPENWINDOW.R** call described below:

```
routine OPENWINDOW.R given XLO, XHI, YLO, YHI, TITLE,
     MAPPING.MODE yielding WINDOW.PTR
```

The parameters XLO, XHI, YLO, and YHI specify the size and position of the window with respect to the computer screen. These coordinates are INTEGERS in the range 0..32767. The point (0,0) defines the lower left-hand corner of the screen, and (32767,32767) is the upper right-hand corner. Window size and position specifications *include* title bar, border and menu bar, (a window whose YHI is 16383 will NOT overlap a window whose YLO is 16383). The TITLE parameter is of mode TEXT and specifies the window title.

The **MAPPING.MODE** parameter defines how the window contents will appear inside the visible portion of the window. The following modes are available:

MAPPING.MODE = 0:

Contents mapped to largest centered square within window.

MAPPING.MODE = 1:

WORLD.XLO (of world coordinate system set by **SETWORLD.R**) is mapped to the left border of the window, **WORLD.YLO** is mapped to the bottom border, and **WORLD.XHI** is mapped to the right border. The top portion of the world coordinate space may not be visible depending on window size. This mode is useful when the background you want to display is significantly wider than it is tall.

MAPPING.MODE = 2:

WORLD.XLO is mapped to the left border of window, **WORLD.YLO** is mapped to bottom border, and **WORLD.YHI** is mapped to the top border. The right portion of the world coordinate space may not be visible depending on window size. This mode is useful when the background you want to display is significantly taller than it is wide.

Selecting which window will display your icons, graphs, and forms is accomplished by associating the window with one or more viewing transforms. In this way the **VXFORM.V** variable not only specifies which viewing transform will be used to draw subsequent graphics, but also in which window the graphics appear. It should therefore be noted that objects

drawn under the same **VXFORM.V** value *cannot* appear in two different windows. Viewing transforms are "attached" to a window through the **SETWINDOW.R** call. Set **VXFORM.V** to the desired transformation number, and then call **SETWINDOW.R** given the **WINDOW.ID** of the window that should contain the objects drawn under this transform. For example:

```
  ''-- create two windows, one directly above the other
call OPENWINDOW.R given 8192, 24576, 16383, 32767,
     "Top Window", 1 yielding WINDOW1.PTR
call OPENWINDOW.R given 8192, 24576, 0, 16383,
     "Bottom Window", 1  yielding WINDOW2.PTR

  ''-- attach viewing transform 1 and 2 to the top
  ''-- window, and 3 to the bottom

let VXFORM.V = 1
call SETWINDOW.R given WINDOW1.PTR
let VXFORM.V = 2
call SETWINDOW.R given WINDOW1.PTR
let VXFORM.V = 3
call SETWINDOW.R given WINDOW2.PTR

  ''-- show icon1 in top window and icon2
  ''-- in bottom window
let VXFORM.V = 2
display ICON1 with "icon1.icn" at (16383, 16383)
let VXFORM.V = 3
display ICON2 with "icon2.icn" at (16383,16383)
```

8.2 Setting and Getting the Attributes and Events of a Window

Calling **OPENWINDOW.R** yields a display entity pointer. The **DFIELD.F** routine can then be used to access window *fields*. In addition, a window *control routine* can be defined by the programmer. As with dialog box, menu bar, and palette control routines, the window control routine is called automatically whenever a user performs some action upon the window with the mouse.

8.2.1 Window Attributes or "Fields"

A window display entity has several predefined field names. See table 8-1. The **DFIELD.F** routine is used to get a pointer to the field, while attributes **DDVAL.A, DARY.A,** and **DTVAL.A** can be read or written to the field by your program. Fields with the access code "RW" represent modifiable components of your window. To see the result of a change made to a **DDVAL.A, DARY.A** or **DTVAL.A** attribute you must redisplay the modified field using a DISPLAY statement.

Table 8-1. Window Display Field Names

Field Name	Attribute	Access	Description
WIDTH	DDVAL.A	RW	Current window width in screen space
HEIGHT	DDVAL.A	RW	Current window height in screen space
VIEWWIDTH	DDVAL.A	R	Width of visible portion of NDC space
VIEWHEIGHT	DDVAL.A	R	Height of visible portion of NDC space
TITLE	DTVAL.A	RW	Title displayed at top of window
HSCROLLABLE	DDVAL.A	RW	> 0 if window should have a horizontal scroll bar
VSCROLLABLE	DDVAL.A	RW	> 0 if window should have a vertical scroll bar
HTHUMBSIZE	DDVAL.A	RW	Width of horizontal scroll bar thumb range (0.0 - 1.0)
VTHUMBSIZE	DDVAL.A	RW	Height of vertical scroll bar thumb range (0.0 - 1.0)
HTHUMBPOS	DDVAL.A	RW	Current position of the horizontal scroll bar from left edge, range (0-HTHUMBSIZE)
VTHUMBPOS	DDVAL.A	RW	Current position of the vertical scroll bar from top edge, range (0-VTHUMBSIZE)
PANEWIDTH	DARY.A	RW	Array of integers describing width (in characters) of each pane of the status bar.
STATUSTEXT	DARY.A	RW	Array of text values shown in each status bar pane
XCLICK	DDVAL.A	R	X location of last mouse click (in NDC units)
YCLICK	DDVAL.A	R	Y location of last mouse click (in NDC units)
XMOVE	DDVAL.A	R	Current X location of mouse (in NDC units)
YMOVE	DDVAL.A	R	Current X location of mouse (in NDC units)
BUTTONDOWN	DDVAL.A	R	If nonzero, the mouse button is currently being held down
BUTTON	DDVAL.A	R	Identifies which of the mouse buttons was last pressed
DOUBLECLICK	DDVAL.A	R	If nonzero, the last click was a double click.

For example, to dynamically reset the title on a window, use:

```
let DTVAL.A(DFIELD.F(“TITLE”, WINDOW.PTR)) = “My New Title”
display DFIELD.F(“TITLE”,. WINDOW.PTR)
```

To determine the top of the window canvas after the window has been resized, use:

```
let TOP = DDVAL.A(DFIELD.F("VIEWHEIGHT", WINDOW.PTR))
```

8.3 Window Events

To receive asynchronous notification of window events such as scrolling, resize and close, the programmer must set up a control routine for the window using the routine **SET.WINCONTROL.R** as follows:

```
call SET.WINCONTROL.R given WINDOW.PTR, CONTROL.ROUTINE
```

where the control routine is formatted as follows:

```
routine CONTROL.ROUTINE given EVENT.NAME, WINDOW.PTR
   yielding BLOCK.DEFAULT
...
Define EVENT.NAME as a text variable
Define WINDOW.PTR as a pointer variable
Define BLOCK.DEFAULT as an integer variable
```

The control routine code should set **BLOCK.DEFAULT** to "1" if the default action is NOT to be performed. The text variable **EVENT.NAME** contains one of the event names shown in table 8-2.

Table 8-2. Event Names

Event Name	Default Action	Affected Fields	Description
CLOSE	Terminate application	None	Sent when user selects window go away icon.
RESIZE	Redraw window contents	WIDTH HEIGHT VIEWWIDTH VIEWHEIGHT	Sent when the user resizes or maximizes the window.
VSCROLL	None	VTHUMBPOS	Sent whenever the user moves the vertical scrollbar thumb.
HSCROLL	None	HTHUMBPOS	Sent whenever the user moves the horizontal scrollbar thumb.
MOUSE-CLICK	None	XCLICK YCLICK BUTTONDOWN BUTTON DOUBLECLICK	Sent whenever any mouse button is pressed down, or lifted up.
MOUSE-MOVE	None	XMOVE YMOVE	Sent whenever mouse movement occurs.

For example, the following control routine can be used to receive some window events:

```
routine WINDOW.CONTROL given EVENT.NAME, WINDOW.PTR
   yielding BLOCK.DEFAULT
...
select case EVENT.NAME
  case "CLOSE"
    write as "Attempt to close window...", /
    let BLOCK.DEFAULT = 1'' dont terminate application

  case "VSCROLL"
    write DDVAL.A(DFIELD.F("VTHUMBPOS", WINDOW.PTR)) *32768 as
     "Window vertically scrolled to ", D(7,2), /
  ...
  default
endselect
end
```

8.4 Scrollable Windows

Scroll bars provide a more natural mechanism for panning across a scene too large to fit inside the boundaries of your window. This is a common condition after *zooming* into a rectangular section of your graphics area. Scroll bars should be added to the window at the time it is created through the **HSCROLLABLE** and **VSCROLLABLE** fields. The following code creates a scrollable window:

```
call OPENWINDOW.R (4096, 28672, 0, 32768, "Scrollable Window", 0)
    yielding WINDOW.PTR
let DDVAL.A(DFIELD.F("HSCROLLABLE", WINDOW.PTR)) = 1
let DDVAL.A(DFIELD.F("VSCROLLABLE", WINDOW.PTR)) = 1
call SET.WINCONTROL.R(WINDOW.PTR, 'CONTROL.ROUTINE')
call SETWINDOW.R(WINDOW.PTR)
```

You can set the width of the scroll bar thumb either before or after the window has been displayed. The **DDVAL.A** attribute of the **HTHUMBSIZE** and **VTHUMBSIZE** fields contains a real number between 0.0 and 1.0. Set this attribute to the percentage of the scroll bar area you wish the thumb to occupy. The size of a scroll bar thumb should represent the *ratio* of viewable area to total area. For example, if the boundaries defined by **SETWORLD.R** are (`w.xlo, w.xhi, w.ylo, w.yhi`), but the *larger* total area occupied by graphics is (`t.xlo, t.xhi, t.ylo, t.yhi`) you should set up the thumb sizes as follows:

```
let DDVAL.A(DFIELD.F("HTHUMBSIZE", WINDOW.PTR)) =
(w.xhi - w.xlo) / (t.xhi - t.xlo)
let DDVAL.A(DFIELD.F("VTHUMBSIZE", WINDOW.PTR)) =
(w.yhi - w.ylo) / (t.yhi - t.ylo)
display DFIELD.F("HTHUMBSIZE", WINDOW.PTR))
display DFIELD.F("VTHUMBSIZE", WINDOW.PTR))
```

Manipulation of the scroll bars by the user will not automatically pan the scene in the window. This action will only send a **HSCROLL** or **VSCROLL** event to the window's control routine informing of the change to the scroll bar thumb position. At this time, the **DDVAL.A** attribute of the **HSCROLLPOS** field will specify the distance from the left-hand side of the horizontal scroll thumb to the left-hand side of the window. **DDVAL.A** of **VSCROLLPOS** is the distance from the top of the window to the top of the vertical scroll thumb. In each case "1.0" is the

total length of the scroll bar. Therefore, these attribute values are in the range [`0.0, 1.0-HTHUMBSIZE`] and [`0.0, 1.0-VTHUMBSIZE`], respectfully. To implement "panning" in the above example you would need the following code in your window control routine:

```
routine WINDOW.CONTROL given EVENT.NAME, WINDOW.PTR
   yielding BLOCK.DEFAULT
...
select case FIELD.NAME
  case "VSCROLL", "HSCROLL"
    let w.xlo = t.xlo + (t.xhi-t.xlo) *
      DDVAL.A(DFIELD.F("HSCROLLPOS", WINDOW.PTR))
    let w.xhi = w.xlo + (t.xhi-t.xlo) *
      DDVAL.A(DFIELD.F("HSCROLLSIZE", WINDOW.PTR))
    let w.ylo = t.ylo + (t.yhi-t.ylo) *
      DDVAL.A(DFIELD.F("VSCROLLPOS", WINDOW.PTR))
    let w.yhi = w.ylo + (t.yhi-t.ylo) *
      DDVAL.A(DFIELD.F("VSCROLLSIZE", WINDOW.PTR))

    call SETWORLD.R(w.xlo, w.xhi, w.ylo, w.yhi)
...
```

8.5 Status Bars

All windows can display a status area at the bottom of the frame called a *status bar.* The status bar is composed of several individual rectangles (panes) of varying width; each containing a line of text. You can define the size of each pane before the window is displayed, and set the text displayed in a pane after the window has been rendered.

Each element of the array pointed to by the **`DARY.A`** attribute of the **`PANEWIDTH`** field specifies the maximum number of characters that can be shown by the corresponding pane. Note that the width of the first status pane is automatically set based on the size of the window. The width specification for the first status pane is always ignored.

Each element of the **`DARY.A`** attribute of the **`STATUSTEXT`** field defines the text to display in the corresponding pane. The first pane is also used to show the status text for a highlighted menu item, or palette button (See chapter 3). A window containing a status bar with three panes could be initialized as follows:

```
'' create the window
call OPENWINDOW.R given 16383, 32768, 8192, 24576,
  "Example Window...", 0 yielding WIN.PTR

'' add a status bar to the window
reserve PANE.WIDTHS(*) as 3
let PANE.WIDTHS(1) = 0   '' not used by SIMGRAPHICS II
let PANE.WIDTHS(2) = 10
let PANE.WIDTHS(3) = 20
let DARY.A(DFIELD.F("PANEWIDTH", WIN.PTR)) = PANE.WIDTHS(*)

'' associate this window with the current viewing transformation.
'' This operation will display the window
call SETWINDOW.R(WIN.PTR)

'' Set the status bar text
```

```
reserve STATUS.TEXT(*) as 3
let STATUS.TEXT(1) = “Pane One”
let STATUS.TEXT(2) = “Pane Two”
let STATUS.TEXT(3) = “Pane Three”
let DARY.A(DFIELD.F("STATUSTEXT", WIN.PTR)) = STATUS.TEXT(*)

‘’ Update display of the status bar text only
display DFIELD.F(“STATUSTEXT”, WIN.PTR)
```

9. Advanced Topics

9.1 Drawing Icons Without SIMDRAW

Icons are specified in SIMGRAPHICS II as arrays of pairs of coordinate values. Each pair of coordinates gives the location of a point in Cartesian space. SIMDRAW produces these arrays automatically. However, they may also be generated manually. The first subscript in a coordinate pair selects either x-coordinates (index = 1) or y-coordinates (index = 2); the second subscript determines a point. For example, the statements:

```
LET SHAPE.ARRAY(1,1) =   40.
LET SHAPE.ARRAY(2,1) = -100.
LET SHAPE.ARRAY(1,2) =   40.
LET SHAPE.ARRAY(2,2) =    0.
```

specify points at (40,-100) and at (40,0).

The coordinate arrays are stored in `REAL` (not `DOUBLE`) mode. This allows a range of model-oriented coordinates and provides sufficient precision for a display device.

Obviously, each coordinate array can only specify an area drawable by an unbroken line. However, complex images can easily be built up using separate arrays for each separate area. These arrays can be referenced in individual subroutines (which include calls to the graphics control routines described in the next section) which are called from the `DISPLAY` routine for the complete icon. Thus, only a single `DISPLAY` routine is required for each icon, however complex; and the individual components of an icon can be controlled separately (e.g., they can change color or size, as required by the application).

9.2 Writing a Display Routine

A display routine is an attribute of a display entity which can be set programmatically. This routine is called automatically by SIMGRAPHICS II whenever it is necessary to display a graphical entity. This routine can contain code to either draw a display entity from scratch using calls to create output primitives, or to modify the display of a display entity created in SIMDRAW. The attribute is called **`DRTN.A`** and is set as follows:

```
Let DRTN.A = 'V.<routine_name>'
```

The heading of the actual routine is defined like this:

```
Display routine <name> given ICON.PTR
define ICON.PTR as a pointer variable
```

The SIMSCRIPT II.5 run-time library routines (described below and in Appendix A) support a number of graphic primitives directly. These routines are called by SIMDRAW. The programmer can also call them directly from a `DISPLAY` routine to draw lines, polygons, circles and arcs, fill delineated areas with color patterns, and write strings of characters. Other primitives allow selection of line style, text sizing, and a variety of colors and fill patterns.

The DISPLAY routine generates a screen image of an icon. It connects the points specified in the coordinate arrays described above, and then adds color, a fillstyle, a line style, and so on, to the image. SIMGRAPHICS II provides a default DISPLAY routine, **DICON.R**. When icons are generated using SIMDRAW, either this default or a user-written DISPLAY routine may be used. When icons are generated without the editor, a user-written routine must be provided. Use of a non-default routine is indicated by assigning a value to the attribute **DRTN.A**:

```
LET DRTN.A(entity) = 'name'
```

DISPLAY routines consist largely of calls on the graphic style routines described below. Other statements may be included to pass values to these routines. All style values remain set until they are changed.

9.2.1 Color

In SIMGRAPHICS II color is specified using an integer value ranging from 0 to 255. This value is an index into a color table whose entries must be initialized programmatically. The routine **GCOLOR.R** defines a color index given the red, green and blue components of the color (color component values range from 0 to 1000). For example, to define index 15 as "green":

```
let RED = 0
let GREEN = 1000
let BLUE = 0
call GCOLOR.R( 15, RED, GREEN, BLUE )
```

Color index number 0 refers to the background color of the window selected through **VXFORM.V**. To set a window's background color to "blue":

```
call GCOLOR.R( 0, 0, 0, 1000 )
```

If your icon has been created by SIMDRAW, you can still reset its color in your program. To enable this feature, from SIMDRAW you must select the primitive whose color you want to be programmatically definable and use the **Edit/Properties** option. From the **Properties** dialog, check the **Define color using DCOLOR.A** check box. The following code will set the color of appropriately defined primitives to "green":

```
let DCOLOR.A(ICON.PTR) = 15
```

9.2.2 Drawing Areas

The primitive operations in this section generate a closed polygon that may be hollow or filled with a solid color, a hatch style, or a pattern. The graphic style routines for areas are called first, and set the appearance of the area.

CALL FILLSTYLE.R(*style*)

Set fillstyle, as follows:

0 = hollow

1 = solid
2 = pattern
3 = hatch

CALL FILLINDEX.R (*index*)
Set pattern or hatch fill selection. Six distinct styles of hatch are available. Hatch styles are as follows:

1 = Narrow spaced diagonal lines
2 = Medium spaced diagonal lines
3 = Wide spaced diagonal lines
4 = Narrow spaced cross hatch
5 = Medium spaced cross hatch
6 = Wide spaced cross hatch

CALL FILLCOLOR.R (*color*)
Set color of solid or hatched area.

CALL FILLAREA.R (*n, points(*)*)
Fill the area, joining the last point to the first point, if necessary, using the present fillstyle and fillcolor.

CALL CIRCLE.R (*points(*)*)
Draw a circle, where ***points*(..,1)** indicates the center, and ***points*(..,2)** is any point on the perimeter.

CALL SECTOR.R (*points(*), rad*)

Draw an arc, where ***points*(..,1)** indicate the center, and ***points*(..,2)** and ***points*(..,3)** are the end points. The sector is drawn counterclockwise from the second to the third points specified. If ***rad*** is not zero, join ends of arc to the center point, and fill.

9.2.3 Drawing Lines

The primitive operations in this section generate solid and dotted lines. The graphic style routines are called first, and set the appearance of the line.

CALL LINESTYLE.R(*style*)
SIMGRAPHICS II supports a number of line styles. The following styles are provided on most implementations:

1 = (solid)
2 = (long dash)
3 = (dotted)
4 = (dash dotted)
5 = (medium dashed)
6 = (dash with two dots)

CALL LINECOLOR.R (*color*)
Color (as described in Chapter 4).

CALL LINEWIDTH.R (*width*)
Width, given in NDC units.

CALL POLYLINE.R (*n, points(*)*)
Joins n points whose x and y coordinates are given in the 2-dimensional real array points(*).

9.2.4 Drawing Points (Markers)

SIMGRAPHICS II supports a primitive operation to mark points on the display surface. The graphic style routines that control appearance are called first.

CALL MARKTYPE.R (*type*)
Where ***type*** is a polymarker type, and where:

1 = dot
2 = cross
3 = asterisk
4 = square
5 = X
6 = diamond

CALL MARKCOLOR.R (*color*)
Polymarker color.

CALL MARKSIZE.R (*size*)
Polymarker size, in NDC units.

CALL POLYMARK.R (*n, points(*)*)
Writes n markers using the current marker type, color, and height.

9.2.5 Direct Character Output

Text can be written directly onto the graphics screen. It is displayed using the text size and color attributes. The output is centered around coordinate points (0, 0), and consequently a modeling transformation can be used to place the information where desired on the display surface.

For example:

```
LET VXFORM.V = 5
CALL SETWORLD.R(0, 79, 0, 23)
```

It may be useful to define a transform to an 80 by 24 coordinate system for writing lines of text, as above. If a text string is to be written, the following call may be used:

CALL WGTEXT.R (*string, x, y*)
Writes string at (x,y) using current text font, color, height, angle, and alignment.

The following graphics routines may be used in display routines to control the appearance of text output:

CALL TEXTFONT.R(font)
Sets font to use. The following fonts are available:

0—SIMGRAPHICS II system font	1—Simple
2—Roman	3—Bold Roman
4—Italic	5—Script
6—Greek	7—Gothic

CALL TEXTCOLOR.R (*color*)
Set color index to use for drawing text.

CALL TEXTSIZE.R (*size*)
Size given in NDC units. Five sizes are available. Exact details depend on the device driver.

CALL TEXTALIGN.R (*horiz, vert*)
Set text alignment, in tenths of degrees from 0 to 360°. The PC implementation only supports 0, 90, 180 and 270°.

CALL TEXTANGLE.R (*degrees*10*)
Set text rotation angles in tenths of a degree.

9.2.6 Character Output Using System Text

Subsequent text primitives created by calls to **WGTEXT.R** can be defined to use a raster text font with a call to **TEXTSYSFONT.R**. i.e.

```
let FAMILY.NAME = "Times Roman"
let POINT.SIZE = 12
let ITALIC.DEGREE = 100'' range is 0-100
let BOLDFACE.DEGREE = 0'' range is 0-100

call TEXTSYSFONT.R given
    FAMILY.NAME, POINT.SIZE, ITALIC.DEGREE, BOLDFACE.DEGREE
call WGTEXT.R("Hello World" X, Y)
```

From above, **FAMILY.NAME** is a string known to the toolkit which identifies the font. Font sizes are in *points*, the size of which is determined by the toolkit. An integer is used to define both the amount of "slant" in the italic, and the darkness of the boldface (for most fonts only two degrees are provided.). Calling **TEXTFONT.R** will re-enable vector fonts.

9.2.7 System Font Browser

A predefined dialog box can be brought up programmatically allowing the user to select system font attributes from those availible on the server. This is done by calling **FONTBOX.R** as follows:

```
let TITLE = "Select a font"

call FONTBOX.R given  TITLE yielding
    FAMILY.NAME, POINT.SIZE, ITALIC.DEGREE, BOLDFACE.DEGREE
```

The yielded arguments are identical to those described above for **TEXTSYSFONT.R**. **FONTBOX.R** will not return until a font has been selected, or *cancel* has been pressed. In this case the result of **FAMILY.NAME** will be "".

9.2.8 Loading a Font Re-definition File

Since the family names for fonts vary from system to system, it is desireable to have a font re-definition file. This file equates a generic family name given in the program to one or more possible system specific names for that font. The format is as follows:

"*generic_name*$_1$" "*system_name*$_{11}$" "*system_name*$_{12}$" "*system_name*$_{13}$" ...
"*generic_name*$_2$" "*system_name*$_{21}$" "*system_name*$_{22}$" "*system_name*$_{23}$" ...
... :
"*generic_name*$_n$" "*system_name*$_{n1}$" "*system_name*$_{n2}$" "*system_name*$_{n3}$" ...

A typical re-definition file could look like this:

```
"times roman" "Times Roman" "Times New Roman" "Times"
```

where *times roman* is the name used within the program. The first font name out of the subsequent re-definition which corresponds to a font available on the system will be used in place of `times roman`. A font file is loaded using the routine **LOAD.FONTS.R** given **FILE.NAME**.

9.2.9 The Shape Example Revisited

This program accomplishes exactly the same function as the program NEWSHAPE discussed in Chapter 6. It is repeated here to illustrate the code which must be included in a model if SIMDRAW is not used. There are situations in which this might be necessary. For example, if some aspect of the icon changes dynamically during the simulation, there is no way to describe that to SIMDRAW. The appearance of the icon could be made dependent on attribute values which change as the simulation proceeds. (We have already seen an example of this in the Gold Mine of Chapter 7.)

```
Preamble   ''Case Study  "OLDSHAPE"
''   This shows a simple dynamic graphics output using SIMGRAPHICS II.
''   It draws a shape and moves it around the screen.

''This version does not use the Icon Editor.
''It shows the details for generating an icon without the Icon Editor.

     Normally mode is undefined
     Processes
        Every SHAPE has
           a SHAPE.ICON
        Define SHAPE.ICON as a pointer variable
     Dynamic graphic entities include SHAPE
     Define .X to mean 1
     Define .Y to mean 2
''   Change SIMGRAPHICS II indices from numbers to words
     Define .RED to mean 2
     Define .GREEN to mean 3
     Define .SOLID.FILL to mean 1
```

```
End ''Preamble

Main
''   Set up the world view and view port
     Let VXFORM.V = 7                              '' View port number
     Call setworld.r(0.0, 2000.0, 0.0, 2000.0)          '' World view
     Call setview.r(0, 32767, 0, 22755)               '' Screen view
''   Reserve the array that describes the ICON and fill it.
     Define ICON.ARRAY as a 2-dim real array
     Reserve ICON.ARRAY as 2 by 7
     Let ICON.ARRAY(.X,1) =   40.  Let ICON.ARRAY(.Y,1) = -100.
     Let ICON.ARRAY(.X,2) =   40.  Let ICON.ARRAY(.Y,2) =    0.
     Let ICON.ARRAY(.X,3) =  100.  Let ICON.ARRAY(.Y,3) =    0.
     Let ICON.ARRAY(.X,4) =    0.  Let ICON.ARRAY(.Y,4) =  100.
     Let ICON.ARRAY(.X,5) = -100.  Let ICON.ARRAY(.Y,5) =    0.
     Let ICON.ARRAY(.X,6) =  -40.  Let ICON.ARRAY(.Y,6) =    0.
     Let ICON.ARRAY(.X,7) =  -40.  Let ICON.ARRAY(.Y,7) = -100.
'' Make 1 second of real time pass for every second of simulated time
     Let TIMESCALE.V = 100
''   Put the process notice for this shape on the event list
''   and associate the icon with it.
     Activate a SHAPE now
     let SHAPE.ICON(SHAPE) = ICON.ARRAY(*,*)
     Start simulation
     Release ICON.ARRAY(*,*)
End ''Main

Process SHAPE
     Define I as an integer variable
''   Set up the parameters for controlling motion
     Let DRTN.A(SHAPE) = 'V.SHAPE'
     Let MOTION.A(SHAPE)   = 'LINEAR.R'
     Let VELOCITY.A(SHAPE) = VELOCITY.F(200.0, PI.C/4)
     Let LOCATION.A(SHAPE) = LOCATION.F(0.0, 0.0)
''   Make the first move
     Work 10 units
''   Change the direction of motion to straight down
     Let VELOCITY.A(SHAPE) = VELOCITY.F(200.0, - PI.C / 2)
     Work 5 units
''   Change the direction of motion again
     Let VELOCITY.A(SHAPE) = VELOCITY.F(200.0, 0.8 * PI.C)
''   Make the shape rotate
     For I = 1 to  60
     Do
          Add PI.C / 60 to ORIENTATION.A(SHAPE)
          Work 0.1 units
     Loop
''   Stop the movement and pause to admire the results
     Let VELOCITY.A(SHAPE) = 0
     Work 5.0 units
End '' SHAPE

Display routine SHAPE Given SHAPE
   Define SHAPE as a pointer variable ''The particular SHAPE to be drawn
    Define .NUMBER.OF.POINTS as an integer variable
    Define ICON.ARRAY as a 2-dim real array
    Let ICON.ARRAY(*,*) = SHAPE.ICON(SHAPE)
    Let .NUMBER.OF.POINTS = dim.f(ICON.ARRAY(1,*))
    Call fillstyle.r(.SOLID.FILL)
    Call fillcolor.r(.RED)
    Call fillarea.r(.NUMBER.OF.POINTS, ICON.ARRAY(*,*))
    Call linecolor.r(.GREEN)
    Call polyline.r(.NUMBER.OF.POINTS, ICON.ARRAY(*,*))
End ''SHAPE
```

9.3 Using Segments

Effective generation of moving or changing display images requires either the complete redrawing of the display at each change or the ability to selectively erase and redraw parts of the display. The first approach requires redundant work in the common case where a few objects are required to move against a static background.

An alternative is to structure the display, identify the grouped components of each object representation, and then provide facilities for manipulating these components. This is done using "segment." Each segment has an identifier and comprises a logical grouping of related graphic primitives. SIMGRAPHICS II provides operations to make a segment visible or invisible or to delete it entirely. Further, by attaching a priority level to each segment, the graphics support can consistently resolve the ambiguity when object representations intersect on the display, i.e. "priority" determines which segment is displayed on top.

A segment provides a means of grouping a related sequence of display primitives (generated by calls to the graphics library routines) and attaching an identifier to the group. The identifier is an integer number returned to the application program when the segment is created, and is usable as a handle to change segment properties (such as visibility).

A program can build segments using one of the following routines:

`CALL OPEN.SEG.R`
Opens a segment. A segment identifier is set in the global variable, **`SEGID.V`**.

`CALL CLOSE.SEG.R`
Closing a segment makes it visible. **`SEGID.V`** is zeroed.

`CALL DELETE.SEG.R (`*`segid`*`)`
Deletes the indicated segment. All primitives in the segment are erased from the display surface.

Only one segment may be open at any time. A segment may not be re-opened and edited. While a segment is open, its ID is available in the global variable **`SEGID.V`**. This value is copied to the **`SEGID.A`** attribute of the display entity upon exit from a `DISPLAY` routine. Note that **`OPEN.SEG`** and **`CLOSE.SEG`** should never be called from within a display routine.

Once a segment is closed, its attributes may be modified using this identifier and the following library routines:

`CALL GPRIORITY.R (`*`segid , pri`*`)`
Explicitly set or change the priority of a segment. ***`pri`*** is an integer in the range 0 to 255.

`CALL GVISIBLE.R (`*`segid , vis`*`)`
Make a segment visible or invisible, where ***`vis`*** is an integer; 0 = invisible, 1 = visible.

`CALL GDETECT.R (`*`segid , det`*`)`
Make a segment detectable to the locator, where ***`det`*** is an integer; 1 = detectable, 0 = not detectable.

CALL GHLIGHT.R (*segid , hi*)

where ***hi*** is an integer; 0 = off, 1 = highlight on. Set the highlighting status of a segment. When highlighted, the entire segment is drawn using color index number 15.

9.3.1 Segment Priority

A segment may have a priority. This priority determines the precedence of any overlapping or intersecting images. A high priority segment is drawn on top of an underlying low priority segment. Priorities are also used to maintain the accuracy of the screen. One image will emerge from behind another unscathed. Segments of zero priority, however, are **not** preserved in this way.

Note that the relationship between differing priorities only exists with segments drawn under the same **vxform.v** value. All segments drawn under one **vxform.v** value will overlap segments drawn under any higher **vxform.v** value, regardless of priority.

When objects overlap, segment priorities determine the order of redrawing moving objects. When priorities are equal, the item drawn last covers anything under it.

When a display routine exits, the value of **SEGPTY.A (*display entity*)** is given to **GPRIORITY.R** to set the priority of the segment. A value of zero for this attribute causes the default priority, zero.

9.3.2 Using Priority Zero

Objects in priority zero are not redrawn when their bounding box is overlapped by moving objects. This makes animation faster. Objects in priority zero will be eaten if they are overlapped by moving objects.

Static objects that will never be crossed or otherwise overdrawn by an animated object may be drawn with priority zero. This is particularly important if the bounding box of the static object is much larger than the object itself and is crossed by animated objects.

Unimportant items crossed by moving objects can often be represented in priority zero. This could leave their image in a temporarily damaged state, but might provide a visual trace of the path of moving objects. For instance, if the entire display surface is cross-hatched in priority zero (using **FILLSTYLE.R**), moving objects will appear to wear paths in the image. A model could then periodically refresh its background to repair the damage.

9.3.3 Other Segment Operations

Library routine **GDEFERRAL.R** determines whether or not the screen is immediately updated as segment status changes. When several overlapping segments are being modified at once, faster operation may result from the following sequence of statements:

```
CALL GDEFERRAL.R(1)
(modify the segments)
CALL GDEFERRAL.R(0)
```

9.3.4 Drawing Backgrounds

Color number zero is the background color and often makes a good background surface. Routine **GCOLOR.R** can be used to make it appropriate to the application. For instance, blue could represent water, green could be grass, and grey or brown could serve for a factory floor.

GCOLOR.R can also be used to change the current representation of a color. For example, the background color could be changed from dark to light to represent the time change from night to day.

9.4 Additional Attributes of [Dynamic] Graphic Entities

Graphical properties can be given only to temporary entities, and not to permanent ones. In addition to the attributes described above and any user-defined attributes, an entity defined as GRAPHIC has the following system-defined attributes:

SEGID.A (*entity*)	Integer, segment identifier
SEGPTY.A (*entity*)	Integer variable. Display priority.
ORIENTATION.A (*entity*)	Angle in radians.
LOCATION.A (*entity*)	Location in world coordinates.

The following constructs can be queried directly to access the X and Y coordinates of the location represented by **LOCATION.A** (actually, functions are invoked to access the values.):

LOCATION.X (*entity*)	X-coordinate of entity location.
LOCATION.Y (*entity*)	Y-coordinate of entity location.

Both **LOCATION.X** and **LOCATION.Y** return values in real world coordinate units. When working with DYNAMIC GRAPHIC entities, the values of **LOCATION.X** and **LOCATION.Y** will change as simulated time advances. They are always up-to-date in SIMSCRIPT event and process routines.

DYNAMIC GRAPHIC entities have the following additional system-defined attributes:

VELOCITY.A (*entity*)	Velocity of the object.
MOTION.A (*entity*)	Subroutine pointer to a subroutine called periodically to animate the object.
CLOCK.A attribute (*entity*)	Double variable; time of last position update. This value is maintained by the routine called through **MOTION.A**.

The following functions are provided to access the X and Y components of the velocity vector represented by **VELOCITY.A**:

VELOCITY.X (*entity*)	X-component of velocity.
VELOCITY.Y (*entity*)	Y-component of velocity.

LOCATION.A and **VELOCITY.A** are left-monitored attributes. Any change in the location of a graphic entity must cause it to be redisplayed. That is why the location must be updated using the **LOCATION.F** function, not by updating the X and Y coordinates separately. The function **LOCATION.F** takes the coordinates and generates a value of the same type as **LOCATION.A**. You must set the values of location and velocity as follows:

```
let LOCATION.A(entity) = LOCATION.F(X_coord,Y_coord)
let VELOCITY.A(entity) = VELOCITY.F(speed, angle)
```

VELOCITY.F works in a similar way for velocity attributes.

9.5 Low-Level Input Constructs

This paragraph covers more elementary interactive input than was covered by the paragraphs on SIMGRAPHICS II forms. Graphical input is provided by the mouse. The following routines apply.

Routine **READLOC.R**, described in Appendix A, returns the location of a mouse click, in real world coordinates. The calling routine gives the following input parameters:

POSX X-coordinate of a position in the window, in real world coordinates.
POSY Y-coordinate of the position.

This point is transformed onto the window surface with a viewing transformation specified by the current value of **VXFORM.V**. The graphical cursor is placed on the display surface at the indicated point. The value of the next argument indicates the representation of the graphical cursor, as follows:

STYLE Style of graphical cursor:

- 0 = **Cross-hair**. Moves as the mouse is moved.
- 1 = **Rubber Band**. Draw a straight line from the initial point to the cursor position, following the moving cursor.
- 2 = **Rubber Box**. Draw a rectangle. One corner is at the initial cursor position, and the other corner follows the cursor.
- 3,4 = If the global variable **DINPUT.V** points to another icon, that icon follows the cursor on the screen, at the location of the mouse.
- 16= **Read locator** asynchronously from within a process routine.

The graphical cursor is under control of the mouse, and remains so until the mouse button signals completion of the graphical input operation.

When the operation is completed, the final cursor position is returned in the three yielding parameters:

NEWX X-coordinate of a position on the display surface, in Real World Coordinates.
NEWY Y-coordinate of the position.

VXFORM.V Viewing transformation in effect at new cursor position.

READLOC.R, called with one of the styles listed above, can be called either from a process or outside of simulation, e.g. from MAIN. If called from a process, it stops the scheduling of all processes. This effectively stops the animation. With a value of STYLE equal to 16, you must call **READLOC.R** from a process, either directly or through a chain of routine calls. Such a call suspends only the process that called **READLOC.R**, while the other processes continue to be scheduled. Otherwise, STYLE=16 behaves like STYLE=3. The suspended process is reactivated when a mouse button is pressed. If the variable **DINPUT.V** is set to point to some icon, its **LOCATION.A** attribute is updated automatically as the locator is moved. **DINPUT.V** is always displayed under viewing transformation number 0.

9.5.1 Selecting a Segment

SIMGRAPHICS II allows the operator to select a specific object from those displayed on the screen. This is done by calling two routines, as follows:

GDETECT.R Marks segments as detectable. Selection applies only to detectable segments.

READLOC.R A side effect of **READLOC.R** is that the identity of the nearest detectable segment is returned in global variable **G.4**. Appendix A describes the rules that determine which segment is nearest.

A special routine is provided to simplify handling of the most common case of selection. Routine **PICKMENU.R**, described in Appendix A, is the simplified menu selector.

9.6 Programmatically Definable System Cursor

The system cursor is usually shown by a small arrow. The default cursor can be changed in your SIMGRAPHICS program as follows:

```
call SETCURSOR.R(1)          '' set to busy (watch) cursor
...
call SETCURSOR.R(0)          '' reset to the normal (arrow) cursor
```

9.7 Time Unit Conversion for Simulation Graphics

Many users of graphics have similar needs regarding time and distance units, and related unit conversions. This section contains information to help centralize solutions to these needs.

The SIMSCRIPT II.5 language provides standard time units of DAYS, with built-in conversion procedures to handle weeks, months, and years, and other conversion procedures to handle **HOURS** and **MINUTES**. The words in the language operate under the assumption that **TIME.V = 1.0 UNITS** means that one day has elapsed in the simulation.

SIMGRAPHICS II provides time synchronization through the variable **TIMESCALE.V**. This variable converts elapsed time (in hundredths of a second) into simulated time units. The operation:

```
LET TIMESCALE.V = 100
```

will cause 1 unit (1 day) of simulation time to operate in every elapsed second, assuming only that the computer can process the model fast enough. Likewise, the operation:

```
LET TIMESCALE.V = HOURS.V * MINUTES.V * 60 * 100
```

will cause exact real time synchronization, with 1 unit of simulation time to operate in every elapsed day.

Many problems work better in different units. For these, SIMSCRIPT provides the undimensioned syntax **UNITS**, which is scaled by the user. The language syntax that describes **MINUTES, HOURS, DAYS, WEEKS, MONTHS**, and **YEARS** is not used in problems worked in **UNITS**.

For example, consider a problem in which it is decided that the basic time unit is the microsecond. It is better to use statements like:

```
WAIT 7 UNITS ''units in microseconds
```

which allows 7 microseconds of simulated time to pass. To observe simulation progress at the rate of 1 microsecond of simulated time every 1 second of elapsed time, use the initialization operation:

```
LET TIMESCALE.V = 100
```

Appendix A. SIMGRAPHICS II Variables and Routines

This appendix describes the routines and variables that are common to all implementations of SIMSCRIPT II.5 SIMGRAPHICS II.

Function ACCEPT.F (FORM.PTR, CNTRL.RTN)

Arguments:

FORM.PTR	A pointer to the graphic input form to be used. This pointer was obtained in the SHOW statement.
CNTRL.RTN	This is either the name of a control routine to control the graphic interaction, or simply 0 to specify no control routine. If there is no control routine, then it is left entirely up to the automatic processing to manage the interaction.
Function:	Accept graphic input from the screen.
Description:	Returns the reference name of the last selected field in a form. Any data which may have been entered by the user is then accessible through the value attributes and names of the various fields in the form.

Routine CIRCLE.R (POINTS(*))

Arguments:

POINTS(*)	Real, 2-dimensional array, reserved as 2 by N, where N ≥ 2. Values are in real world coordinates. **POINTS(1, ...)** are the x-coordinates. **POINTS(2, ...)** are the corresponding y-coordinates.
Function:	Draw a circle.
Description:	A circle is drawn, with the center at the first given point. The second given point is any point on the circumference. Any points after the second are ignored. The circle is drawn with attributes set through **FILLCOLOR.R**, **FILLSTYLE.R**, and **FILLINDEX.R**.

Routine CLEAR.SCREEN.R

Description:	Erases all graphics in the current screen. No segments or entities are destroyed.

Attribute CLOCK.A (DSPLENT)

Mode:	Double.
Subscript:	Pointer to a GRAPHIC entity or to a DYNAMIC GRAPHIC entity.

Description: Time of last position update. This value is maintained by the routine called through **MOTION.A**.

Routine CLOSE.SEG.R

Side effects: The value of **SEGID.V** is set to zero.

Function: Close a segment.

Description: The segment is closed. No additional primitives may be added to it. Its representation is made up-to-date on the display surface. No drawing occurs until the segment is closed.

Routine CLOSEWINDOW.R (WINDOW.ID)

Arguments:

WINDOW.PTR Pointer. Identifier returned by **OPENWINDOW.R**

Description: Closes a SIMGRAPHICS II window given its pointer. Note that graphical entities contained in this window are NOT destroyed.

Attribute DARY.A (FIELD.PTR)

Arguments:

FIELD.PTR A pointer to a field in a graphic input form.

Description: An array of text variables from a field on an input form. For instance, in list boxes it is a pointer to the array of text variables in the list.

Attribute DDVAL.A (FIELD.PTR)

Arguments:

FIELD.PTR A pointer to a field in a graphic input form.

Function: Access the numeric value attribute of a field.

Description: This is used to accept or alter information in one field of a form. For instance, in value boxes it is the value which the user entered or which was pre-set.

Routine DELETE.SEG.R (SEG.ID)

Arguments:

SEG.IDInteger. Identifier of a segment, as produced by **OPEN.SEG.R**.

Function: Delete a segment.

Description: The segment is deleted. Its representation is removed from the display surface. Space occupied by its data structures is recycled.

Function DFIELD.F (FIELD.NAME, FORM.PTR)

Arguments:

FIELD.NAME The text string name of a graphic field.

FORM.PTR A pointer to a graphic input form.

Function: Returns a pointer to the specified field.

Description: The acquired field pointer is used to access the attributes of the graphic input field, for examining input, altering values, or setting control attributes.

Attribute DRTN.A (DSPLYENT)

Mode: Subprogram variable. The subprogram does not return a value.

Subscript: Pointer to a GRAPHIC entity or to a DYNAMIC GRAPHIC entity.

Function: Associates a display routine with an instance of an entity.

Description: The use of a particular display routine is indicated through the value of the **DRTN.A** attribute. The display routine is normally generated by the compiler and has a name of the form **'V.routine_name'**.

Attribute DTVAL.A (FIELD.PTR)

Arguments:

FIELD.PTR A pointer to a graphic input field.

Function: Access a text value associated with the field.

Description: **DTVAL.A** is used to access a text value associated with a field. For instance, in text boxes, **DTVAL.A** has the value of the input or pre-set text.

Routine FILEBOX.R(FILTER, TITLE) yielding PATH.NAME, FILE.NAME

Arguments

FILTER String. This variable can either be a wild card, or a full or partial file name that uses wildcards.

TITLE String. The title of the file selection dialog box.

PATH.NAME String. The path to the file selected by the user.

FILE.NAME String. The name of the file selected from the dialog box.

Function: Displays the standard dialog box for browsing through the directory structure.

Routine FILLAREA.R (COUNT, POINTS(*))

Arguments:

COUNT Integer. Number of points to process.

POINTS(*) Real, 2-dim array, reserved as 2 by N, where N ≥ COUNT. Values are in real world coordinates. **POINTS(1, ...)** are the x-coordinates. **POINTS(2, ...)** are the corresponding y-coordinates.

Function: Draw a line or a polygon.

Description: A filled area (polygon) is drawn connecting the indicated points. The area is drawn in the current fillcolor, fillindex, and fillstyle specified through routines **FILLCOLOR.R, FILLINDEX.R**, and **FILLSTYLE.R**. If the last point is not the same as the first, they will be connected to close the filled area.

Routine FILLCOLOR.R (COLOR.INDEX)

Arguments:

COLOR.INDEX Integer. Color index number. May have values from 0 to 255.

Function: Set color of subsequent fill areas.

Description: See the discussion of colors elsewhere in this report for further information.

Routine FILLINDEX.R (INDEX)

Arguments:

INDEX Integer. Identifies a style of fill hatch:

1 = narrow diagonals
2 = medium diagonals
3 = wide diagonals
4 = narrow crosshatch
5 = medium crosshatch
6 = wide crosshatch

Function: Set style of subsequent fill hatch areas.

Description: The standard configuration offers six fill hatch styles.

Routine FILLSTYLE.R (STYLE)

Arguments:

STYLE Integer. Identifies a style of fill:

0 = Hollow area

1 = Solid color

2 = Pattern (appearance is device-dependent)

3 = Use hatch fill. Pattern is set by FILLINDEX.R.

Function: Set style of subsequent fill areas.

Description: The standard configuration offers the indicated styles.

Routine FONTBOX.R(TITLE) yielding FAMILY.NAME, POINT.SIZE,BOLDFACE.DEGREE ITALIC.DEGREE

Arguments:

TITLE String. The label for the font dialog box.

FAMILY.NAME String. The return of the font name selected in the dialog box.

POINT.SIZE Integer. The size of the font selected in points.

ITALIC.DEGREE Integer. Return value of the font slant selected by the user. The range is from 0 to 100. For most fonts only two values are allowed.

BOLDFACE.DEGREE
Integer. Return value of the "boldness" of the font. The range is from 0 to 100. For most fonts only two values are allowed.

Function: Provides a predefined dialog box for font specification that can be brought up programatically to allow the user to select system font attributes.

Routine GCOLOR.R (COLOR.INDEX, RR, GG, BB)

Arguments:

COLOR.INDEX **COLOR.INDEX** is an integer with values from 0 to 255.

RR Integer. Amount of red to use, 0 to 1000.

GG Integer. Amount of green to use, 0 to 1000.

BB Integer. Amount of blue to use, 0 to 1000.

Function: Set a color representation for subsequent use under the indicated color index. **RR,GG**, and **BB** are the portions of red, green, and blue to use. The effect is seen as objects are redrawn.

Routine GDEFERRAL.R (DEFER)

Arguments:

DEFER Integer.

1 = set deferral on
0 = set deferral off

Function: Set deferral status of entire system.

Description: When deferral is on, system changes may be made without updating the display. When deferral is off, changes to the display will be seen immediately.

For example, when a number of possibly overlapping segments are deleted, response may be faster if deferral is on before deletion, and is then turned off afterwards.

Routine GDETECT.R (SEG.ID, DETECT)

Arguments:

SEG.ID Integer or integer. A segment ID value as returned by **OPEN.SEG.R**.

DETECT Integer.

0 = set undetectable status
1 = set detectable status

Function: Make a segment detectable or not.

Description: A detectable segment can be detected using **READLOC.R** or **PICKMENU.R**.

Routine GHLIGHT.R (SEG.ID, HIGHLIGHT)

Arguments:

SEG.ID Integer. A segment ID value as returned by **OPEN.SEG.R**.

HIGHLIGHT Integer.

0 = normal display
1 = highlighted

Function: Set the highlighting status of a segment. A highlighted segment draws attention to itself on the display surface.

SIMGRAPHICS II implements highlighting by drawing the entire segment using the color index number.

Routine GPRIORITY.R (SEG.ID, PRIORITY)

Arguments:

SEG.ID Integer. A segment ID value as returned by **OPEN.SEG.R**.

PRIORITY Integer. Range is 0 to 255. The entire graphical display is composed by drawing segments in the order of their priority, starting with priority zero. This implies that, if segments overlap, the segment with the higher priority overwrites the segment with lower priority. For segments of the same priority, the drawing order is undefined. Deleting a segment automatically redraws all segments with bounding boxes intersecting the bounding box of the deleted segment, but not segments with priority zero.

Function: Set or change the priority of a segment.

Routine GUPDATE.R

Description: Draws all unsegmented primitives.

Routine GVISIBLE.R (SEG.ID, VISIBLE)

Arguments:

SEG.ID Integer. A segment ID value as returned by **OPEN.SEG.R**.

VISIBLE Integer.

0 = Set invisible status
1 = Set visible status

Function: Make a segment visible or invisible.

Routine HANDLE.EVENTS.R(WAIT.FOR.EVENT)

Arguments:

WAIT.FOR.EVENT

Integer.

0—Return immediately
1—Wait for a mouse event to occur.

Description: Routine to handle low-level toolkit events such as window resizing. Necessary for tight loop constructs occupying a large amount of time.

Routine LINEAR.R (DSPLYENT)

Arguments:

DSPLYENT Pointer to a DYNAMIC GRAPHIC entity.

Function: Manage one object with linear motion.

Description: The values of **LOCATION.A** (present location) and **CLOCK.A** (time of last change) are updated, and the entity displays itself. The user does not call this routine. It is automatically assigned as the motion attribute of a DYNAMIC GRAPHIC entity.

Routine LINECOLOR.R (COLOR.INDEX)

Arguments:

COLOR.INDEX Integer. **COLOR.INDEX** is an integer with values from 0 to 255.

Function: Set color of subsequent lines.

Description: See the discussion of colors elsewhere in this report for further information.

Routine LINESTYLE.R (STYLE)

Arguments:

STYLE Integer. The following style values are supported:

1 = solid
2 = long dash
3 = dotted
4 = dash dotted
5 = medium dashed
6 = dash with two dots
7 = short dash

Function: Set style of subsequent lines.

Routine LINEWIDTH.R (WIDTH)

Arguments:

WIDTH Integer. In NDC units. The typical range is 00 to 32767 NDCs.

Function: Set width of subsequent lines.

Routine `LISTBOX.SELECTED.R (LISTBOX.PTR, INDEX) Yielding SELECTED`

Arguments:

`LISTBOX.PTR` Pointer to a listbox `FIELD` within a form.

`INDEX` Integer. Index into array of list items

`SELECTED` Return value. `1` if item has been selected, `2` if it has been double-clicked on, `0` otherwise.

Description: Given a listbox field pointer and an index into the array of items, this routine returns whether this item is currently selected or has been double-clicked.

Routine `LOAD.FONTS.R(FILE.NAME)`

Arguments:

`FILE.NAME` String. The name of the file to be loaded.

Description: Loads the font re-definition file **`FILE.NAME`**. A font re-definition file defines equivalent names for font families. For example, a program may use the font name Times, when on Windows systems the equivalent system font is Times New Roman and on Unix systems it is Times Roman. Then the font re-definition file would consist of the line:

```
"Times""Times New Roman""Times Roman"
```

The first entry is the generic (program) name and the subsequent entries are the equivalent system fonts.

Left Monitoring Routine `LOCATION.A (DSPLYENT)`

Arguments:

`DSPLYENT` Pointer to a graphic entity, dynamic or static.

Function input value: A pointer to a **`LOCATION.E`** entity. The value must be obtained from **`LOCATION.F (x, y)`**. This value indicates the location of the origin of the object, in real world coordinates.

Function: Provide location for modeling transformation.

Description: Set or change the location of a moving object. Draw or redraw the object if and as necessary. Assignment to this attribute triggers redisplaying of the graphic entity. If you also want to change **`ORIENTATION.A`**, do it before assignment to this attribute. If only **`ORIENTATION.A`** is to be changed, the object should be explicitly redisplayed.

Function LOCATION.F (X, Y)

Arguments:

X	Real, in real world coordinates.
Y	Real, in real world coordinates.

Function value: Pointer to a **LOCATION.E** entity. This entity is constructed from the x and y values to represent a coordinate position, and should only be used in an assignment to **LOCATION.A**.

Function: Set a present location given x and y.

Function LOCATION.X (DSPLYENT)

Arguments:

DSPLYENT	Pointer to a graphic entity, dynamic or static.

Function value: Real, in real world coordinates. This is a read-only value.

Function: Inquire the present X position.

Function LOCATION.Y (DSPLYENT)

Arguments:

DSPLYENT	Pointer to a graphic entity, dynamic or static.

Function value: Real. In real world coordinates. This is a read-only value.

Function: Inquire the present Y position.

Routine MARKCOLOR.R (COLOR.INDEX)

Arguments:

COLOR.INDEX	Integer. **COLOR.INDEX** is an integer with values from 0 to 255.

Function: Set color of subsequent markers.

Description: See the discussion of colors elsewhere in this report for further information.

Routine MARKSIZE.R (SIZE)

Arguments:

SIZE	Integer. The value is 0 to 32767, in NDC units.

Function: Set size of subsequent markers.

Routine MARKTYPE.R (TYPE)

Arguments:

TYPE Integer. Identifies a marker type. Permitted values include:

1 = dot
2 = cross
3 = star
4 = square
5 = X
6 = diamond

Function: Set type of subsequent markers.

Routine MESSAGEBOX.R (MESSAGE.TEXT, TITLE.TEXT)

Arguments:

MESSAGE.TEXT Text. Identifies a one line message.

TITLE.TEXT Text. Title displayed in title bar of message.

Function: Display a one-line message to the user.

Description: A modal dialog box containing one **OK** button and a one line message will be displayed. The user must click on the **OK** button before execution can resume.

Attribute MOTION.A (DSPLYENT)

Mode: Subprogram variable. The subprogram does not return a value.

Subscript: Pointer to a DYNAMIC GRAPHIC entity.

Function: Provides an animation velocity management routine.

Description: The use of a particular animation velocity management routine is indicated through the value of the **MOTION.A** attribute. The default routine is named **'LINEAR.R'**.

Routine MSCALE.R (FACTOR)

Arguments:

FACTOR Scale factor, DOUBLE.

Function: Set the scaling component of the system modelling transformation.

Description: The effect of this routine is reset upon entry to a DISPLAY routine, or with an explicit call of **MXRESET.R** with argument zero, a call to this routine with the argument equal to zero, or before the display of an icon. The scaling factor will take effect only if called from within a display routine or before a call to **CLOSE.SEG.R**.

Routine MXLATE.R (POSX, POSY)

Arguments:

POSX Real. Distance to move.

POSY Real. Distance to move.

Function: Modelling transformation, translation from within a display routine or before a call to **CLOSE.SEG.R**.

Description: Specifies translation (movement) component of a modeling transformation. The translation is cumulative with previous translations. All rotation specified through **MZROTATE.R** is performed before translation.

Will only take effect if called.

Routine MXRESET.R (DSPLYENT)

Arguments:

DSPLYENT Pointer to a graphic entity. An argument of 0 resets all the components of the system's modeling transformation to null.

Function: Reset the modelling transformation to that of the given object.

Description: The rotation is set from **ORIENTATION.A(OBJECT)**. The translation is set from the **LOCATION.A** attribute of the given graphic entity.

Routine MZROTATE.R (THETA)

Arguments:

THETA Real value of rotation, in radians. Positive values indicate counterclockwise rotation.

Function: Modeling transformation, rotation.

Description: Specify rotation component of a modeling transformation. Successive calls on this routine are cumulative. The given rotation is added to previous rotations.

Will only take effect if called from within a display routine or before a call to **CLOSE.SEG.R**.

Routine OPEN.SEG.R

Side effects: Changes the value of global variable **SEGID.V**.

SEGID.V Integer. Identifier of a new segment.

Function: Open a new segment.

Description: A new graphic segment is opened and made able to accept graphic primitive operations.

Routine OPENWINDOW.R (XLO, XHI, YLO, YHI, TITLE, MAPPING yielding WINDOW.PTR

Arguments:

XLO Integer. NDC coordinate for left edge of window (with respect to screen)

XHI Integer. NDC coordinate for right edge of window (with respect to screen)

YLO Integer. NDC coordinate for bottom edge of window (with respect to screen)

YHI Integer. NDC coordinate for top edge of window (with respect to screen)

TITLE Text. Title of window

MAPPING Integer. Mapping mode of window (0=LCS, 1=X major, 2=Y major)

WINDOW.PTR Pointer to a window display entity.

Description: Opens up a SIMGRAPHICS II window of the prescribed dimensions on the screen and returns a display entity for it. **SETWINDOW.R** can then be used to associate a viewing transformation to this window. The MAPPING flag defines how NDC space is mapped to the four boundaries of the window.

Attribute ORIENTATION.A (DSPLYENT)

Mode: Real, in radians. Positive values specify counterclockwise rotation.

Subscript: Pointer to a GRAPHIC entity or to a DYNAMIC GRAPHIC entity.

Description: Sets orientation of a graphic entity, for the modelling transformation. When **ORIENTATION.A** is used, it should be set before a value of **LOCATION.A** is set.

Routine PICKMENU.R GIVEN ARRAY(*) YIELDING INDEX

Arguments:

ARRAY(*) 1-dim POINTER array. Each element of the array is a graphic entity pointer.

INDEX Integer. Subscript to array produced by **PICKMENU.R**.

Function: Selection from a menu using the mouse.

Description: Waits for the user to make a selection using the mouse. Reads the position of the cursor when the user completes the selection. If the cursor is outside the bounding box of any of the passed entities, sets index to zero. Otherwise, returns the index of the highest priority object with the bounding box containing the point where the mouse was clicked.

Routine `POLYLINE.R (COUNT, POINTS(*))`

Arguments:

`COUNT`	Integer. Number of points to process.
`POINTS(*)`	Real, 2-dimensional array, reserved as 2 by N, where N ≥ `COUNT`. Values are in real world coordinates. **`POINTS(1, ...)`** are the x-coordinates. **`POINTS(2, ...)`** are the corresponding y-coordinates.

Function: Draw a line or a polygon.

Description: A line is drawn connecting the indicated points. The line is drawn with the current line color, line style, and line width, as set by calling **`LINECOLOR.R`**, **`LINESTYLE.R`** and **`LINEWIDTH.R`**. If the last point is the same as the first, this line will close to form a polygon.

Routine `POLYMARK.R (COUNT, POINTS(*))`

Arguments:

`COUNT`	Integer. Number of points to process.
`POINTS(*)`	Real, 2-dimensional array, reserved as 2 by N, where N ≥ `COUNT`. Values are in real world coordinates. **`POINTS(1, ...)`** are the x-coordinates. **`POINTS(2, ...)`** are the corresponding y-coordinates.

Function: Draw a series of markers.

Description: Markers are drawn at the indicated points. The markers are drawn in the current markcolor, marksize, and marktype, provided through routines **`MARKCOLOR.R`**, **`MARKSIZE.R`**, and **`MARKTYPE.R`**.

Routine `POSTSCRIPTCTRL.R(ENABLE, SHOWICON)`

Arguments:

`ENABLE`	Integer. Enable conversion of window to PostScript.
`SHOWICON`	Integer. If the value is greater than 0 the conversion button will be displayed in the top-right corner of the window.

Description: Enables and configures PostScript output.

Routine POSTSCRIPT.R(PSFILE, PSSIZE, PSBORDER, PSMONO, PSINVERT,PSHATCH, PSDIALOG)

Arguments:

PSFILE	Text. The name of the output file.
PSSIZE	Real. Height and width of the output in inches.
PSBORDER	Integer. Show a window border in the output.
PSMONO	Integer. Not yet implemented.
PSINVERT	Integer. Not yet implemented.
PSHATCH	Integer. Not yet implemented.
PSDIALOG	Integer. Bring up a dialog box to get options for the conversion to PostScript.

Description: Converts all graphics in the current window to PostScript.

Routine PRINT.SEG.R given SEGMENT.ID, USE.DIALOG yielding SUCCESS

Arguments:

SEGMENT.ID	Integer. Segment identifier.
USE.DIALOG	Integer. If **USE.DIALOG** is nonzero the system print dialog is displayed allowing the user to set print options.
SUCCESS	Integer. Nonzero if printing was completed.

Description: Prints a portion of a window.

Routine PRINT.WINDOW.R given WINDOW.PTR, USE.DIALOG yielding SUCCESS

Arguments:

WINDOW.PTR	Pointer to a window display entity. Returned from **OPENWINDOW.R.**
USE.DIALOG	Integer. If **USE.DIALOG** is nonzero the system print dialog is displayed allowing the user to set print options.
SUCCESS	Integer. Nonzero if printing was completed.

Description: Prints a window.

Routine READ.GLIB.R (FILE.NAME)

Arguments:

FILE.NAME	Text. File name of the graphics library.

Function: Read a graphics library file from disk.

Description: This routine will read a graphics library file created by SIMDRAW into SIMGRAPHICS II. Subsequently, all icons, forms and graphs contained in the library can be accessed through the SHOW and DISPLAY statements. Note that the file **graphics.sg2** is automatically read in during SIMGRAPHICS II initialization (if it exists).

Routine READLOC.R (POSX, POSY, STYLE) YIELDING NEWX, NEWY, XFORM.V

Arguments:

POSX Real, in real world coordinates: X anchor point of the cursor.

POSY Real, in real world coordinates: Y anchor point of the cursor.

STYLE Integer:

0 = Draw only cursor while reading

1 = Draw rubber band while reading

2 = Draw box while reading

3,4 = allows a global variable **DINPUT.V** to be assigned a pointer to a SIMGRAPHICS II graphic entity which will be repeatedly updated with a new **LOCATION.A** value, thus tracking the locator until the locator read is terminated.

16 = may be used within a SIMSCRIPT process routine (which **READLOC.R** will suspend). The locator position will be sampled from the timing mechanism, allowing the locator to be active while a simulation is running. A suspended process is reactivated after the mouse is clicked.

NEWX Final X position of the mouse in real world coordinates.

NEWY Final Y position of the mouse in real world coordinates.

XFORM.V Value of the viewing transformation used to map NDC locator position into real world coordinates.

Function: Location function, using the mouse.

Description: The graphic cursor is started at the given (**POSX,POSY**). The operator moves the cursor as desired, and the cursor is tracked by hardware or software. **READLOC.R** scans the viewing transformations in reverse numerical order - from 15 to 0 - until the NDC position can be reverse-transformed. If the locator is within a viewport specified by some transformation number, this number is returned. In this way, movement of the mouse between viewports or menus may be detected and acted upon. As a side effect, the global integer **G.4** will be set to the ID of the selected segment.

Function RGTEXT.F (X, Y, MAXLEN)

Arguments:

X (ignored)
Y (ignored)
MAXLEN (ignored)

Function: Read graphic text.

Description: A text string is read in from a popup dialog box and returned.

Routine SEARCH.GLIB.R yielding ARRAY.OF.ITEMS

Arguments:

ARRAY.OF.ITEMS
1 dimensional text array. An array containing the names of objects in any loaded graphic library.

Description: Returns an array containing names of all objects in any loaded graphics libraries. The array should NOT be released.

Routine SECTOR.R (POINTS, FILL)

Arguments:

POINTS(*)

Real, 2-dimensional array, reserved as 2 by N, where N ≥ 3. Values are in real world coordinates. **POINTS(1, ...)** are the x-coordinates. **POINTS(2, ...)** are the corresponding y-coordinates.

FILL Integer. Identifies filling procedure:

0 = Draw an arc of a circle using current line style and color
1 = Draw a sector of a circle, fill with current fill style and fill color

Function: Draw an arc or a sector of a circle.

Description: The first point identifies the center of a circle. The second point, any point on the circumference, is the beginning of the arc. An arc is drawn counterclockwise to the third point. Any points after the third are ignored.

Routine SEG.BOUNDARIES.R (SEGMENT.ID)
yielding SEG.XLO, SEG.XHI, SEG.YLO, SEG.YHI

Arguments:

SEGMENT.ID Integer. Identifies a segment.

SEG.XLO Integer. Left side of bounding box in NDC units.

SEG.XHI Integer. Right side of bounding box in NDC units.

SEG.YLO Integer. Bottom side of bounding box in NDC units.

SEG.YHI Integer. Top side of bounding box in NDC units.

Function: Compute the bounding box of any existing segment.

Description: Computes the NDC coordinates defining the bounding rectangle of the segment given by **SEGMENT.ID**. Can be called before the segment has been made visible.

Left Monitoring Routine SEGID.A (DSPLYENT)

Arguments:

DSPLYENT Pointer to a graphic entity.

Function input value: Integer. A segment identifier as produced by **OPEN.SEG.R**.

Function: Utility operation. Removes image of a segment and causes a new image to be drawn.

Description: An assignment to this attribute will delete the previous segment, if one exists. Assigning the value 0 to this attribute will erase the representation of an object.

Assignment to this attribute has the following side-effects:

1. If the old value is not zero it is taken to be a segment identifier of an existing segment. That segment is deleted.
2. If the new value is not zero, it is taken to be a segment identifier of an existing segment. That segment is displayed with modeling transformation determined by **LOCATION.A** and **ORIENTATION.A.**

Global Variable SEGID.V

Mode: Integer. Segment identifier.

Description: While a segment is open, its ID is available in the global variable **SEGID.V**. This value is copied to **SEGID.A** upon exit from a DISPLAY routine.

When a segment is closed, the value of **SEGID.V** becomes zero.

Attribute SEGPTY.A (DSPLYENT)

Mode: Integer. Display priority.

Subscript: Pointer to a GRAPHIC entity or to a DYNAMIC GRAPHIC entity.

Description: The priority of displays of graphic entities is supplied through this attribute. Segments with a higher priority are displayed in front of lower-priority segments. The order of displaying segments of equal priority is not defined.

Priority 0 is treated specially by SIMGRAPHICS II. Segments of this priority are not automatically redisplayed by the system. This feature can be used to make animation faster.

Routine SET.ACTIVATION.R (FORM.PTR, ACTIVATE)

Arguments:

FORM.PTR Pointer to any form or form field.

ACTIVATE Integer:

0 = Deactivate or "grey out" the field.

1 = Activate the field; make it selectable.

Function: Set activation state of a form or field.

Description: Sets the activation state of a field on a form. A deactivated field will appear “greyed out,” i.e., it is visible but cannot be interacted with. Setting the activation state of a dialog box or menu bar will apply that state to all fields contained therein. Fields are initially activated.

Routine SETCURSOR.R(CURSORSTATUS)

Arguments:

CURSORSTATUS Integer:

0 = Set the cursor to the normal (arrow) cursor.

1 = Set the cursor to the busy (watch) cursor.

Function: Sets the cursor status to busy or normal and changes its iconic representation to a watch (hourglass) or arrow.

Routine SET.LISTBOX.TOP.R(LISTBOX.PTR, TOP.INDEX)

Arguments:

LISTBOX.PTR Pointer. Pointer to list box field obtained from **DFIELD.F**.

TOPINDEX Integer. Index of the list item to appear in the top of the list window.

Description: Will force the identified list box item to appear at the top of the list box window by appropriately scrolling the list box.

Routine SETVIEW.R (V.XLO, V.XHI, V.YLO, V.YHI)

Arguments:

V.XLO Integer, in Normalized Device Coordinates.

V.XHI Integer, in Normalized Device Coordinates, where (0 ≤**V.XLO** < **V.XHI** ≤32767).

V.YLO Integer, in Normalized Device Coordinates.

V.YHI Integer, in Normalized Device Coordinates, where (0 ≤ **V.YLO** < **V.YHI** ≤32767).

On the standard SIMGRAPHICS II configuration, the visible viewing surface includes all points (x, y) where 0 ≤ x ≤ 32767 and 0 ≤ y ≤ 32767. The parameters given to **SETVIEW.R** should define a rectangle within the visible viewing surface.

Function: Set viewport on the display surface.

Description: This function defines a rectangular viewport region on the display surface. Areas, lines, and points outside this region are clipped.

Routine SET.WINCONTROL.R given WINDOW.PTR, CONTROL.ROUTINE

Arguments:

WINDOW.PTR Pointer to the window display entity.

CONTROL.ROUTINE
Name of the routine invoked on a window event.

Function: Invoke the given CONTROL.ROUTINE on any of the following asynchronous window events: CLOSE, RESIZE, VSCROLL, HSCROLL, MOUSECLICK, MOUSEMOVE.

Routine SETWINDOW.R (WINDOW.PTR)

Arguments:

WINDOW.PTR Identifier for a SIMGRAPHICS II window returned by **OPENWINDOW.R**

Description: Associates the current viewing transform (**VXFORM.V**) to the window with the given id. This means the all subsequent drawing to the viewing transform will appear in this window. This allows the programmer to use **VXFORM.V** to specify which window will receive subsequent graphic input. Note that a single viewing transform cannot be drawn in two separate windows. Therefore, this call must be used if graphics are to be drawn in more than one window.

Routine SETWORLD.R(W.XLO, W.XHI, W.YLO, W.YHI)

Arguments:

W.XLO	Real. In real world coordinates.
W.XHI	Real. In real world coordinates, where (**W.XLO ne W.XHI**).
W.YLO	Real. In real world coordinates.
W.YHI	Real. In real world coordinates, where (**W.YLO ne W.YHI**).

Note: Usually **W.XLO<W.XHI** and **W.YLO<W.YHI**. However, **SETWORLD.R** can be used to invert or mirror-image a transformation.

Function: Defines a square or rectangle in world space. The argument values define the area to be displayed. Points outside this area are clipped, and are not displayed.

Description: Sets the mapping of problem-oriented coordinates, given in real world coordinates. These coordinates are converted into device-oriented coordinates, given in NDC units, through parameters given to **SETVIEW.R**.

This operation is applied to all points after the modeling transformation is specified by calling **MXRESET.R, MXLATE.R**, or **MZROTATE.R**.

SETWORLD.R allows the entire coordinate system to be upside down or reversed.

Routine SYSTIME.R YIELDING CURRENT.TICK

Arguments:

CURRENT.TICK	Integer. The value represents the elapsed time, since midnight, in 1/100 second on most systems.

Description: This routine returns the current time of day in the indicated units.

Routine TEXTALIGN.R (HORIZ, VERT)

Arguments:

HORIZ	Integer. Value = 0, 1, or 2.
VERT	Integer. Value = 0, 1, 2, 3, or 4.

Function: Set portion of character that is aligned upon the graphic text position.

Description: The standard configuration supports the following values:

0 = left or bottom justified (the default)
1 = center justified
2 = right or top justified
3 = bottom of character cell

4 = top of character cell

The character cell extends both above and below the actual character.

Routine TEXTANGLE.R (ANGLE)

Arguments:

ANGLE Integer. Selects an angle in tenths of degrees, 0 to 3600.

Function: Set display baseline for subsequent characters.

Description: This operation allows displaying characters at angles.

Routine TEXTCOLOR.R (COLOR.INDEX)

Arguments:

COLOR.INDEX Integer. **COLOR.INDEX** is an integer with values from 0 to 255.

Function: Set color of subsequent characters.

Routine TEXTFONT.R (FONT)

0—SIMGRAPHICS II system font
1—Simple
2—Roman
3—Bold Roman
4—Italic
5—Script
6—Greek
7—Gothic

Arguments:

FONT Integer. Indicates which font to use.

Function: Set text font of subsequent characters.

Routine TEXTSIZE.R routine (SIZE)

Arguments:

SIZE Integer. The character height in NDC units, with a range of 0 to 32767.

Function: Set size of subsequent characters.

Description: Each operating environment may support different sizes for text. The actual size of the displayed characters will be the size closest to the requested value.

Routine TEXTSYSFONT.R given FAMILY.NAME, POINT.SIZE, ITALIC DEGREE, BOLDFACE.DEGREE

Arguments:

FAMILY.NAME String. **FAMILY.NAME** is a string known to the toolkit which identifies the font.

POINT.SIZE Integer. The size of the font in points.

BOLDFACE.DEGREE
Integer. An integer that denotes the darkness of the font. Range is 0 to 1000.

ITALIC.DEGREE
Integer. An integer that denotes the slant of the font. Range is 0 to 1000.

Description: Used to set the system font. If called, the font set using **TEXTFONT.R** is temporarily ignored.

System Global Variable TIMESCALE.V

Mode: Integer.

Description: Scales Real time (1/100 second) per simulated time unit.

System Global Variable TIMESYNC.V

Mode: Subprogram variable.

Description: When non-zero, this subprogram variable will point to a user exit routine, called with the following parameters:

TIME.PROPOSED

GIVEN argument; mode is DOUBLE. The value will be greater than **TIME.V**.

TIME.COUNTERED

YIELDING quantity; mode is DOUBLE. The user must set this to a value between **TIME.V** and **TIME.PROPOSED**.

The YIELDING parameter will be taken as the next simulated time.

When events or processes are scheduled or canceled by the user time exit routine, the value returned for **TIME.COUNTERED** must be less than **TIME.PROPOSED**. This causes a rescan of the time file, preventing potential difficulties.

The user exit routine reached through the **TIMESYNC.V** variable is called whenever the simulated clock is to be updated, but before any animation is performed.

Left Monitoring Routine VELOCITY.A (DSPLYENT)

Arguments:

DSPLYENT Pointer to a DYNAMIC GRAPHIC entity.

Function input value: A pointer to a **VELOCITY.E** entity . This value indicates the velocity of the object, in real world coordinate units per simulated time units.

Assigning a value of 0 to **VELOCITY.A** causes the object's position updates to cease. This stops the object from moving.

Except for the special value of 0, the value of **VELOCITY.A** can only be set to the function value produced by **VELOCITY.F (speed, theta).**

Function: Associate a constant velocity with a dynamic graphic entity.

Description: Set or change the velocity of a moving object. Draw or redraw the object if necessary.

Function VELOCITY.F (V, THETA)

Arguments:

V Real. Velocity in real world coordinate units per simulated time units.

THETA Real. Angle of motion, in radians.

Function value: Pointer to a **VELOCITY.E** entity. This entity is constructed from the velocity and angle values to represent a vector location.

Function: Set a present velocity given absolute velocity and angle.

Description: Returns the indicated function value.

Function VELOCITY.X (DSPLYENT)

Arguments:

DSPLYENT Pointer to a DYNAMIC GRAPHIC entity.

Function value: Real. This function returns the x-coordinate of the current velocity of the object, in real world coordinate units per simulated time units. This is a read-only value.

Function: Inquire the present velocity, in the X direction.

Function VELOCITY.Y (DSPLYENT)

Arguments:

DSPLYENT	Pointer to a DYNAMIC GRAPHIC entity.

Function value: Real. This function returns the y-coordinate of the current velocity of the object, in real world coordinate units per simulated time units. This is a read-only value.

Function: Inquire the present velocity, in the Y direction.

Variable VXFORM.V

Mode: Integer. Value between 0 and 15, inclusive.

Subscript: None.

Function: Indicates which viewing transformation is in effect.

Description: The default transformation is provided by **VXFORM.V = 0**, a one-for-one mapping of real world coordinates into Normalized Device Coordinates.

A mapping is indicated by the value of the system variable **VXFORM.V**. This variable indicates which transformation is to be defined, redefined, or used.

Multiple mappings between real-world spaces and areas on the display screen are supported by SIMGRAPHICS II. Such user-defined mappings are specified with **VXFORM.V** and are defined through parameters given to subroutines **SETWORLD.R** and **SETVIEW.R**. (Used in conjunction with **SETWINDOW.R** to define which window receives subsequent graphic input and output.)

Routine WGTEXT.R (STRING, X, Y)

Arguments:

STRING	Text.
X	Real, in real world coordinates.
Y	Real, in real world coordinates.

Function: Write a text string.

Description: The text string is written starting at the indicated point. The string is written in the current text alignment, text angle, text color, text size, and text font, using values provided through the routines that set these properties.

Appendix B. Conversion to SIMGRAPHICS II

B.1 What is SIMGRAPHICS II?

SIMGRAPHICS II is the next generation in graphics support for SIMSCRIPT II.5. It inherits all of the capabilities of SIMGRAPHICS I but provides more features, functions, and performance than its predecessor.

The user interface capabilities of SIMGRAPHICS I have been greatly improved upon in SIMGRAPHICS II. Dialog boxes are no longer composed of output primitives drawn in the canvas of a window, but are shown with the standard toolkit dialogs available on the particular machine. For example, Microsoft Windows SIMSCRIPT applications will bring up Windows menus and dialog boxes as well. Therefore, features that come with these toolkits (such as editable text boxes, pinup menus, and movable dialog boxes) can now be employed by SIMSCRIPT applications.

A new graphics editor has been written for SIMGRAPHICS II: SIMDRAW. This editor lets you to see your icons, graphs and forms displayed in the same window. The new editor makes selecting, grouping, and changing styles, colors, and modes much easier than in SIMGRAPHICS I. And since this editor is written in SIMGRAPHICS II, it also takes advantage of the toolkit support on a given computer, giving it the look and feel of the environment under which it is running.

All icons, graphs, and forms created in the SIMGRAPHICS II editor are contained in one file (**`graphics.sg2`**). This means that directories will no longer be clogged up with **`.icn`**, **`.grf`**, and **`.frm`** files. A conversion utility (**`simcvt`**) is provided with SIMGRAPHICS II that can convert existing icon, graph and form files to the SIMGRAPHICS II format.

Some additional graph types are available in SIMGRAPHICS II. A *digital clock* type of graph is available which is used in the same manner as the existing SIMGRAPHICS I analog clock. A new type of graph used for displaying scalar values is the *digital display* which shows a single scalar value surrounded by a titled box. A *text meter* graph is also available for SIMGRAPHICS II which can be used show display text variables. Charts and trace plots have added capabilities. Multiple data sets within a chart can be displayed with their bars shown side-by-side, as well as stacked or on top of each other. Trace plots can now have data compressed in the X direction, rather than being lost when time exceeds the maximum X value. A discrete flag can be checked in the editor to cause consecutive points in a trace plot to be connected by a discrete line rather than by linear interpolation.

B.2 Differences Between SIMGRAPHICS I and II

Converting your program to run under SIMGRAPHICS II may require some source code changes. Since SIMGRAPHICS II uses the vendor toolkit to display dialog boxes and menu bars, some of the SIMGRAPHICS I capabilities are not supported due to vendor toolkit restrictions. In addition to the possible source code changes, you must convert your

icon, graph, and form files to the SIMGRAPHICS II format. This is done using **simcvt**. The specific differences between SIMGRAPHICS I and II are outlined below.

B.2.1 Icons

Under SIMGRAPHICS II, the priority of an icon is always with respect to other icons drawn under the same viewing transformation. All icons drawn under a lower **VXFORM.V** value will overlap icons drawn under a higher **VXFORM.V,** regardless of priority. This prevents viewports from becoming intertwined. In addition, calling **SETWORLD.R** and **SETVIEW.R** causes everything already drawn under the current **VXFORM.V** value to be redisplayed under the new viewing transformation, i.e. pan and zoom are now automatic.

B.2.2 Graphs

Some minor repositioning may be needed in the conversion due to differences in the implementation.

B.2.3 Forms

In SIMGRAPHICS I, a form may consist of an arbitrary collection of icons, menu bars, check boxes, etc. A form in SIMGRAPHICS II must consist of one of the following three objects:

1. A dialog box containing check boxes, push buttons, text boxes, value boxes, list boxes, radio buttons, and text labels.
2. A menu bar containing menus which in turn contain menu items.
3. A palette (not supported in SIMGRAPHICS I).

These are the only type of display entities that can be passed to an **ACCEPT.F** function.

SIMGRAPHICS I supports four independent options which can be applied to every form and field: terminating, non-pickable, hidden, and user-defined. In SIMGRAPHICS II, only push buttons can be terminating. Every form/field is pickable or activated by default. Any form or field (i.e menu items, menu bars, dialog boxes, check boxes) can be activated or deactivated at runtime using the routine **SET.ACTIVATION.R**. This routine takes the display entity and an integer specifying activation status as arguments. If a 0 is passed, the form or field will be grayed-out by the toolkit. If 1 is passed the form/field will be redisplayed normally. The hidden and user-defined attributes are no longer supported.

B.2.4 Menu Bars

The **DIVAL.A** attribute of a menu bar should not be set in SIMGRAPHICS II. Menus will always go away when the user is done with the interaction.

B.2.5 Dialog Boxes

In SIMGRAPHICS II a dialog box cannot contain icons. Unlike menu bars, there can be more than one dialog box displayed asynchronously at a given time.

B.2.6 Push Buttons

Push buttons can be terminating or nonterminating. In addition, a push button can be a "verify" attribute, which means that every value box within the same dialog box is verified when this button is pushed. The control routine will not be called when a verify button is pushed unless every value box has been successfully verified.

B.2.7 Radio Buttons

A new type of object called a *radio box* is now the container of a group of radio buttons that are logically associated with each other. (i.e. the radio box is a field of the form, and the radio buttons are fields of the radio box). Radio buttons must always be positioned in a column.

B.3 Using the Conversion Utility

SIMCVT is a conversion program that converts your SIMGRAPHICS I (SG1) forms/ graphs/ icons to SIMGRAPHICS II (SG2) format and puts them into the file **`graphics.sg2`**. The **`graphics.sg2`** file will contain the SIMGRAPHICS II objects corresponding to the form/graph/icon files. The SIMGRAPHICS II object receives the name of the SIMGRAPHICS I file.

The files to be converted can be specified on the command line, in a special conversion file list, or interactively. While SIMCVT is converting, it writes messages to the text window, as well to a conversion log file **`simcvt.log`**. All error and warning messages from the conversion will appear in this file.

Note that SIMCVT cannot be considered a "hands-off" conversion utility. Because SIMGRAPHICS II must comply with the underlying window system's philosophy, not all things that were possible in SIMGRAPHICS I will be portable to SIMGRAPHICS II. Therefore, some hand-editing of the converted forms/graphs/icons may be necessary.

B.3.1 Calling SIMCVT — Command Line Arguments

SIMCVT can be called in three different ways: With command line arguments, with a conversion file list argument and without any arguments, in which case you will be prompted for the files to be converted. On Windows, SIMCVT is generally called from within SimLab through the **`Tools`** menu. Command line arguments can be given in a dialog box.

In all cases, the results are written to a file **`graphics.sg2`** in the current directory (where SIMCVT runs), and any error or warning messages will be written to a file **`simcvt.log`**. The different ways to call SIMCVT are:

1. **Files specified as command line arguments:** The given files are converted. **Wildcards are not supported on Windows!** When the conversion is complete, SIMCVT will ask you to press **Return** (to give you time to look at any messages) and exit. For example:

   ```
   simcvt input.frm results.frm
   ```

   On Windows: Specify the arguments **`input.frm`** and **`results.frm`** in the **Arguments** dialog box when you call SIMCVT from the SimLab **Tools** menu (**Convert SG1 to SG2**).

2. **Files specified in a conversion list file:** When you have a list of files to be converted, you may pass this list file as a command line argument to SIMCVT with an `@` prefix. All files whose names are listed in this conversion list file will be converted. This is especially useful on Windows, which does not support wild cards on command lines (you cannot simply enter **`simcvt *.frm`**). For example: In a DOS window, create and edit the conversion list file. Then call SIMCVT with this file. For example, at the DOS command line, type:

   ```
   dir *.frm /w > cnvfiles.lst
   ```

   This creates a file **`cnvfiles.lst`** with the names of all the files with a **`.frm`** extension. Then edit **`cnvfiles.lst`** so that only the wanted names are left.

   Then when calling SIMCVT, specify the command line argument **`@cnvfiles.lst`**.

3. **No command line arguments given:** When no command line arguments are given at all, you will be prompted to enter the file names interactively. Again, wildcards are not allowed here. Enter "**`end`**" to exit SIMCVT.

B.3.2 Possible Problems with Forms

Graphs and icons will convert one-to-one from SIMGRAPHICS I (SG1) to SIMGRAPHICS II (SG2). Only forms can cause some confusion, since the underlying window systems do not support some of the features used by forms in SG1. The features that can no longer be supported under SG2 are listed later in this appendix. Here we will just discuss a few issues related to form conversion:

Controls not contained within a dialog box: When a form (**`*.frm file`**) is converted that contains controls that are not contained by a dialog box, a dialog box will be automatically created for them. In SG2, a control must always be within a dialog box.

B.3.3 A Menu Bar Within a Form

SIMCVT will convert only menu bars that are by themselves in a form file. When you have a menu bar *within* a form (dialog box), SIMCVT will *ignore the menu bar and all controls behind it* and give you an appropriate warning message (conversion for that form stops when the menu bar is found). You should use SimEdit to save the menu bar into a separate form file by itself and then convert it separately.

Any other icons contained in a form cannot be converted directly to SG2. Again, the underlying window systems do not allow mixing icons and controls in a dialog box. Therefore, these icons are converted and then stored in a SG2 object named after the form, but with a **`.icn`** extension. Thus, the icons contained in **`ground.frm`** would be stored in **`ground.icn`**.

Special command line options for form conversion: SIMCVT has two special command line options that control the conversion of forms:

`-i` causes icons found in a form to be ignored rather than being put into a **`*.icn`** SG2 object.

`-v` ***`button_id`***
causes all buttons with the field ID equal to the given **`button_id`** to be set verifying. This option can be used any number of times.

Getting a SG1 DUMP: If you are getting surprising results from SIMCVT, you should use SimEdit to create a **dump** of the form you want to convert. This dump will list all the items contained in the form and give you more information to interpret the output.

B.3.4 Conversion of Files from PC DOS SIMSCRIPT

Problems can arise when trying to use or convert forms/graphs/icons created under PC DOS SIMSCRIPT. This is because PC DOS SIMSCRIPT used an old binary format that is incompatible with the newer SIMSCRIPT releases (all other platforms). Before using SIMCVT to convert your forms/graphs/icons to SIMGRAPHICS II, convert your forms/graphs/icons to ASCII form *using the PC DOS SIMSCRIPT Graphics editor*. This ASCII format can be read without problems.

B.3.5 Miscellaneous Notes

Upper/lower case of file names: The converted forms/graphs/icons are stored in the file **`graphics.sg2`** under the file name they had under SIMGRAPHICS I. All given file names are stored **in lower case** in **`graphics.sg2`**. The `SHOW` command used to show a SIMGRAPHICS form/icon/graph again maps all given file arguments to lower case. This way, the upper/lower case spelling of former file names does not matter.

B.3.6 Features No Longer Supported in SIMGRAPHICS II

The following guidelines should help you to write code that is fully portable between SIMGRAPHICS I and SIMGRAPHICS II. All restrictions are due to the fact that the underlying windowing systems take control over display, use and appearance of menus and dialog boxes. If you follow these guidelines your applications should be able to run under SIMGRAPHICS I and SIMGRAPHICS II without change.

1. **Do not** rely on being able to **change the color of controls** in dialog boxes, i.e. do not make the color of buttons, and boxes significant.

In SIMGRAPHICS II, activation and deactivation of menu items (indicated by color change in SIMGRAPHICS I) will be provided by a call to a routine **`SETACTIVATION.R`**.

2. **Do not** rely on being able to **directly select menus**, i.e. the top menu bar. In SIMGRAPHICS I the control routine is called when you click on the top menu bar and when you select a menu item, the control routine is called again.

   In SIMGRAPHICS II, you can only select menu *items*, i.e. you will only return from an **`ACCEPT.F`** call after you have selected a *leaf* of the menu tree.

3. **Do not** rely on being able to **attach icons to dialog boxes**. In SIMGRAPHICS II, you can, of course, have any number of icons on the screen (and react to clicks there). They just cannot be part of a dialog box.

4. **Do not** rely on **`GCOLOR.R`** to **change the color of icons that have already been drawn**. SIMGRAPHICS I manipulated the color index table of the graphic display it was running on directly. Thus, changing the RGB values of a particular color index immediately changed all the icons drawn with that color index.

   In the supported windowing systems, colors are considered a *resource* that must be allocated and *used afterwards* (this is done by SIMGRAPHICS II). Thus, **`GCOLOR.R`** can affect only the icons drawn *after* its call. The only exception is the background **color**. Changes of color index `0` (background) take immediate effect.

5. **Do not** rely on being able to control the particular menu that is displayed, i.e. do not use the attribute **`DIVAL.A`**.

   In SIMGRAPHICS II, menus are under the control of the windowing system (see guideline number 2).

Index

Numerics

2-D plots 45

A

ACCEPT.F 80-86, 145
ACCUMULATE statement 71
adding an object 33
adding an object to the library 33
align and distribute, shape 42
animation 101
arcs 39
Areas 132

B

bar graph 45, 48
bitmaps 40
bmp 26, 40, 95
button 54
button field 31, 86, 95

C

cascadeable menus 26
center point 42
changing the name of an object 33
chart 45
check box 55
CIRCLE.R routine 133, 145
circles 11, 38
CLEAR.SCREEN.R routine 145
clipboard 36, 54
CLOCK.A attribute 140
clocks 50
CLOSE.SEG.R routine 138, 146
CLOSEWINDOW.R routine 146
color 44
color palette 35, 45
combo box 31, 52, 57, 80, 90
command line arguments 34
continuous surface 48
controls 52, 54, 61
coordinate space boundaries 43
copy option 36
create 49, 52, 60, 71, 80
cut option 36

D

DARY.A attribute 146
data set 48
DDVAL.A 86
DDVAL.A attribute 146
delete option 36
DELETE.SEG.R routine 146
DESTROY command 87
DESTROY statement 110
DFIELD.F function 124, 147, 163
dial 51
dialog box 11, 39, 53, 59
dialog box, tabbed 61
Dialog Editor 31, 52
dials 51
digital display 52
dimension 43
discrete surface 48
DISPLAY command 72, 86
DISPLAY routine 106, 131, 156
DISPLAY statement 72, 107
DRTN.A attribute 131, 147
DTVAL.A attribute 19, 147
DYNAMIC GRAPHIC entities 102, 108, 140, 145

E

edit graphic images 11
editing objects 32
editor, palette 66
ERASE 1, 72, 86, 110

F

field attributes 81, 84
field identifier 80, 85
File Selection Dialog 93, 147
FILEBOX.R routine 147
fill style 46, 49, 161
FILLAREA.R routine 148
FILLCOLOR.R routine 148
FILLINDEX.R routine 148
FILLSTYLE.R routine 149
flip and rotate tools 42
FONTBOX.R routine 149
form pointer 83
forms 88, 123, 141, 171

G

GCOLOR.R routine 149
GDEFERRAL.R routine 150
GDETECT.R routine 150
GHLIGHT.R routine 150
gotolink f 60
GPRIORITY.R routine 151
Graph Editor 12, 31, 44
graph types 12, 31, 71, 171
graphics.sg2 ..... 1, 5, 71, 95, 173
graphs 11
grid 1, 43, 61
group 11
GUPDATE.R routine 151
GVISIBLE.R routine 151

H

HANDLE.EVENTS.R routine 151
histograms 45, 49, 117

I

icons 1, 5, 6, 62, 71, 79, 101, 110, 131, 172
Image Editor 31-36
images 40

K

keyboard accelerators 31

L

label 57
Layout Editor 12, 31, 33
Layout/Group option 40
level meter 51
LINEAR.R routine 152
LINECOLOR. routine 152
lines 11
LINESTYLE.R routine 152
LINEWIDTH.R routine 152
list box 56
LISTBOX.SELECTED.R routine 153
listing objects 33
LOAD.FONTS.R routine 153
LOCATION.A routine 153
LOCATION.F function 154
LOCATION.X function 154
LOCATION.Y function 154

M

making a duplicate of an object 33
MARKCOLOR.R routine 154
MARKSIZE.R routine 154
MARKTYPE.R routine 155
menu bar 62
Menu Bar Editor 31
menu item 65
menus 11
MESSAGEBOX.R routine 155
mode palette 36
modeling transformation 105
MOTION.A routine 155
MSCALE.R routine 155
multi-line text box 56
MXLATE.R routine 156
MXRESET.R routine 156
MZROTATE.R routine 156

N

normalized device coordinates 43

O

OPEN.SEG.R routine 157
OPENWINDOW.R routine 157
ORIENTATION.A routine 157

P

palette button 69
Palette Editor 31
palette separators 69
palette, color 35, 45
palette, mode 35, 66
palette, style 35, 44
pan 43
paste option 36
PICKMENU.R routine 157
pie chart 49
polygons 11, 38
POLYLINE.R routine 158
polylines 37
POLYMARK.R routine 158
PostScript 23, 94
POSTSCRIPT.R routine 159
POSTSCRIPTCTRL.R routine 158
presentation graphics 1, 21, 31, 71, 113
primitives 37
PRINT.SEG.R routine 159
PRINT.WINDOW.R routine 159
priority 41, 113, 138, 139
push buttons 172

R

radio box 56
radio buttons 56, 80, 88, 96, 172, 173
raster file, importing 40
READ.GLIB.R routine 159
READLOC.R routine 160
representation 48
resize 8, 12, 21, 33, 36, 53, 59-68, 125
RGTEXT.F function 161
rotate 42
rounding 38
Running SIMDRAW 31

S

save 11, 15, 24, 31, 32, 71, 174
scroll 43, 56, 62, 79, 89, 123, 127
SEARCH.GLIB.R routine 161
sector 38
SECTOR.R routine 161
SEGID.A routine 162
SEGID.V global variable 162
SEGPTY.A attribute 162
selecting, moving, and resizing 36
SET.ACTIVATION.R routine 163

SET.LISTBOX.TOP.R routine 163
SET.WINCONTROL.R routine 164
SETCURSOR.R routine 163
SETVIEW.R routine 164
SETWINDOW.R routine 164
SETWORLD.R routine 165
snap 1, 43
stacking order 41, 42
style palette 35, 44
system font browser 93
system text 39, 135
SYSTIME.R routine 165

T

Tab Field 60
Tabbed Dialog 60, 87
tables 58
TALLY statement 1, 71
terminating fields 98
text box .17-18, 31, 50-57, 69, 80-88, 96, 147, 171
text meter 52
text primitives 39
TEXTALIGN.R routine 165
TEXTANGLE.R routine 166
TEXTCOLOR.R routine 166
TEXTFONT.R routine 166
TEXTSIZE.R routine 166
TEXTSYSFONT.R routine 167
time trace plot 49
TIMESCALE.V system global variable 167
TIMESYNC.V system global variable 167
trace plots 31, 71, 76, 171
transferring a menu or menu item 64

V

value attributes 84, 145
value box 55
vector text 39
VELOCITY.A routine 168
VELOCITY.F function 168
VELOCITY.X function 168
VELOCITY.Y function 169
verify button 54, 85, 89, 173, 175
vertices defining a primitive 41
viewing transformations 103
VXFORM.V variable 169

W

WGTEXT.R routine 169
window events 126, 127
WITH clause 72

X

X-axis 46
xwd 23, 26, 40, 95

Y

Y-axis 47

Z

zoom 43, 62