```
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 AIRPLANE

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

SIMSCRIPT Graphics User's Manual

CACI

If there are questions regarding the use or availability of this product, please contact CACI at any of the following addresses:

For product Information contact:

CACI Products Company
1011 Camino Del Rio South, suite 230
San Diego, California 92108
Telephone: (619) 542-5224
www.caciasl.com

CACI Worldwide Headquarters
1100 North Glebe Road
Arlington, Virginia 22201
Telephone (703)841-7800
www.caci.com

For technical support contact:

Manager of Technical Support
CACI Products Company
1011 Camino Del Rio South #230
San Diego, CA 92108

Telephone: (619) 542-5224

simscript@caci.com

CONTENTS

FIGURES

TABLES

1. Introduction to SIMSCRIPT Graphics

The goal of a 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 elements of the system. In SIMSCRIPT, animated graphics called *icons* 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 also 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 is prepared with built in language features allowing the programmer to represent entities ranging from aircraft on runways to messages in a communications network.

Icons are custom built graphical objects usually composed of colored circles, lines polygons, and text. They can be used to represent anything from single moving objects to a complicated static background. An icon can even contain JPEG images of any dimension. Using SIMSCRIPT Studio's icon editor, a template of the icon is created, named, and saved to the "graphics.sg2" file. Through the SIMSCRIPT "Show" and "Display" statements, your program can load copies of the icon into display entities.

Effective visualization of statistical quantities may also be important. A time-elapsing simulation sometimes requires that dials, level meters, 2-d plot chars, histograms and pie-charts be dynamic. This allows the user to changes to monitored quantities as they take place. SIMSCRIPT provides several predefined *graph* objects for viewing single quantities in dials, level meters, or histograms. 2-d charts and pie charts allow a whole set of data points to be viewed. Graphs are updated automatically as data quantities change.

Most modern applications allow more user interaction than in the past. Users expect to be able to control the whole application through a menu-bar displayed at the top of the window, or to use a smaller context menu to dynamically make changes to an individual item. Dialog boxes are used as a convenient way to communicate data quantities to a program. Applications will also include palettes, scroll bars, and allow the user to double-click or drag graphical object around the window. SIMSCRIPT provides support for *forms* which include menu bars, dialog boxes, tool bars, and popup message boxes. SIMSCRIPT supports these objects using JAVA which allowing programs to be recompiled and executed on different operating systems without recoding or retooling the graphical user interface.

The SIMSCRIPT Studio development environment provides a point and click interface for creating objects falling into all three of the above categories of *icons*, *graphs* and *forms*. SIMSCRIPT provides both language and runtime support allowing these objects

to be displayed in your program. The objects are contained in a graphics library file called *graphics.sg2* shown as a component in your SIMSCRIPT Studio project tree. The names of all graphical objects can be seen by clicking on the (+) next to "graphics.sg2". Any existing object can be shown in a separate editor by double clicking on its name. You can add new objects to the library by right-clicking on "graphics.sg2" then selecting "new" from the popup menu.

Each chapter in this document explains a different facet of SIMSCRIPT graphics. Chapter 2 describes icons in detail – how to create them in SIMSCRIPT Studio, and how to use them in the program. Chapter 3 outlines the use of segments and display routines to create graphics by program code only. Chapter 4 explains how to make use of SIMSCRIPT's built in chart, graph and monitoring support. In Chapter 6, we move away from the canvas drawn graphics and toward user interface support by describing the use of dialog boxes. Components such as buttons, text and value boxes, list boxes, and tables are studied along with their interaction with both the code and the user. Chapter 7 is devoted to menu-bars with emphasis on how to create and use them in your program. Chapter 8 explains the use of palettes and toolbars. Chapter 9 includes topics related to windows. For example, how to create multiple windows, use scrollbars to implement a pan and zoom, show a status bar, and write code to receive notification of mouse, scrollbar and window manipulation events caused by user interaction.

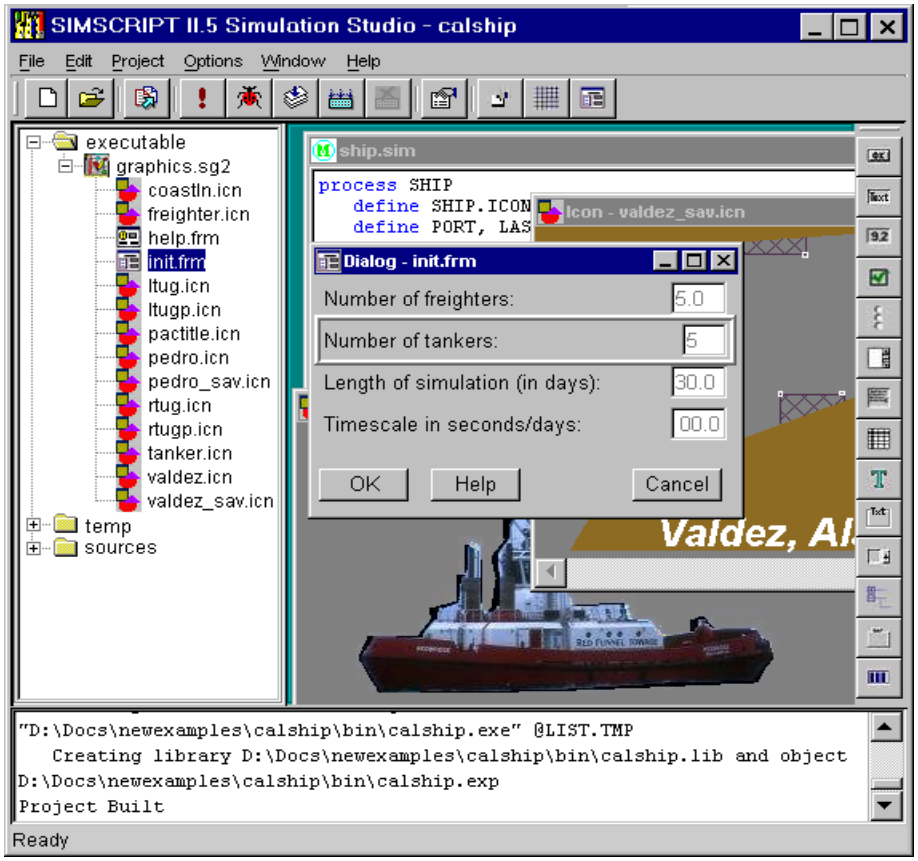

Figure 1-1: SIMSCRIPT Studio showing dialog box, icon, and source code editing.

2. Icons

Icons are used by SIMSCRIPT II.5 to graphically represent any moving or static object inside a window. Icons are typically composed of a group of shapes such as lines, polygons, etc. but can include text and even JPEG images. An icon can be arbitrarily complex. SIMSCRIPT Studio provides an *icon editor* that allows these icons to be defined using a simple and easy to use drag and drop interface. An icon can also be defined entirely by your program. Icons are accessed by your SIMSCRIPT application using the DISPLAY and SHOW statements.

2.1 Getting Started: Adding a Simple Icon

To create a new icon, right-click on the graphics library shown in the contents pane (typically "graphics.sg2") and select "New" from the popup menu. Type a name for your icon into the dialog box. Choose this name carefully since it must be provided to the `SHOW` or `DISPLAY` statement in your program. Usually icon names end in ".icn". Now select "Icon" from the drop down list and click the "Create" button.

When an icon editor window is active a toolbar will appear along the right edge of SIMSCRIPT Studio called the "Icon toolbar". Clicking on any of the buttons in this toolbar will display a row of buttons that can be used to select current fill style, dash, font, color or mode.

Example 2.1: Create a rectangle icon
Create a new icon called "icon1.icn" and add a rectangle to it. To do this, first click on the top button in the Icon editor toolbar. Now select the "rectangle" tool (third button from the left). Click in two different locations in the canvas to define the corners of a rectangle. Save your changes using the File/Save menu.

2.2 Getting Started: Showing the Icon

The simplest way to show an icon in your SIMSCRIPT program is by using the `display` statement:

```
Display <icon_pointer> [with <icon_name> [at (<x>, <y>)]]
```

Where `<icon_pointer>` is your local variable of type pointer, `<icon_name>` is the same name used by the icon editor, and `<x>` and `<y>` define where in the window the icon is to be placed. (NOTE: The "with" and "at" clauses are optional).

Example 2.2: Show the rectangle in a program

In the next part of this example, we will create a small SIMSCRIPT program to display the icon created in Example 2.1.

```
Main
Define ICON.PTR as a pointer variable
Display ICON.PTR with "icon1.icn" at (16383.0, 16383.0)
Read as /
End
```

Note that in the simplest case, you can define a pointer variable to hold your icon. SIMSCRIPT allows processes and temporary entities to be declared as icons in your Preamble. See "Declaring Icons as Entities".

2.3 Using the Icon Editor

The Icon toolbar is located on the right side of SIMSCRIPT Studio's window. The toolbar contains 6 buttons that allow you to create new shapes, change the style and color of existing shapes, change the font of any selected text, or even zoom in and out. Clicking on a button will pop-up another row of buttons.

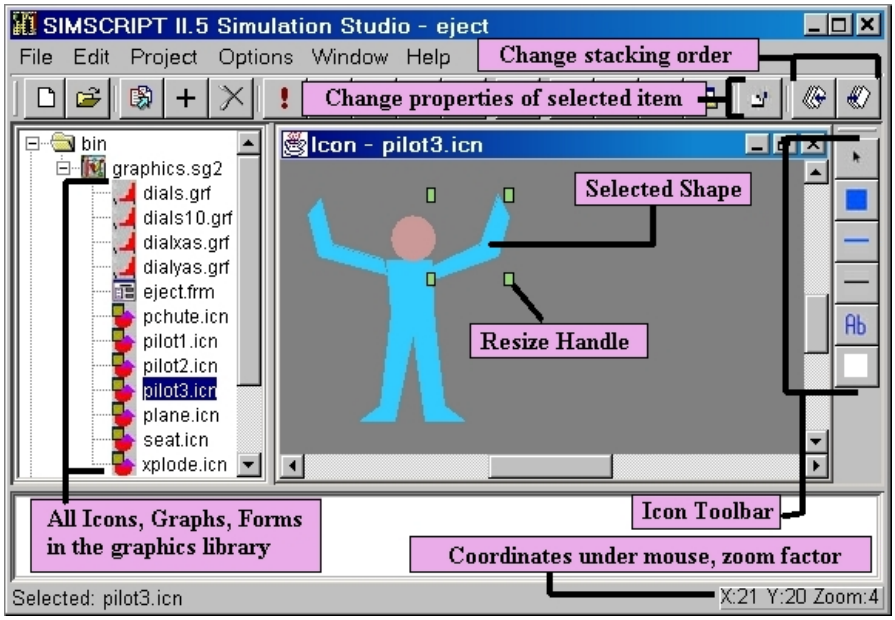

Figure 2-1: Icon editor in SIMSCRIPT Studio

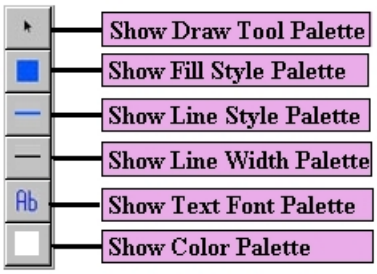

Figure 2-2:The Icon Toolbar

2.3.1 Draw Tool Palette: Creating New Shapes

The *draw tool palette* is displayed by clicking on the top button in the icon toolbar (Initially shown with a small arrow pointing north-west). The palette allows you to create any of the following shapes: Rectangles, filled polygons, polylines, circles, sectors, arcs, filled regions, or text (see figure 2.3). To add a new shape to your icon, click on its button in the mode palette then click in the Icon's edit window. Rectangles and circles require 2 clicks to define the size and shape. You must click three times to define an arc or sector. Shapes such as polylines or polygons require multiple clicks to define the points. When creating a polyline or polygon, double-click to terminate a sequence of line segments.

2.3.2 Draw Tool Palette: Zoom In and Out

The draw tool palette also contains a buttons that will allow you to zoom in and out. This is handy when editing a very small icon. By clicking on the magnifying glass, you enter "zoom" mode. When in zoom mode you zoom by positioning the mouse of the area of interest then clicking with the left button. Clicking with the right mouse button will zoom out. Click on the "Arrow" button to leave zoom mode.

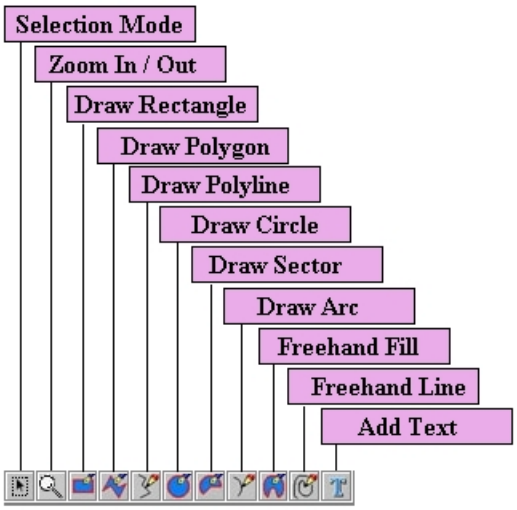

Figure 2-3:The Draw Tool Palette

2.3.3 Fill Palette: Specifying the Fill Style of a Shape

This palette will allow you to change the hatch pattern used to fill rectangles and polygons. To change style, first select any polygon in the edit window. Click on the second button from the top (Blue square) to expose the fill style palette then select one of the styles.

2.3.4 Dash and Width Palettes: Specifying Line Style

The third and fourth buttons from the top of the toolbar allow you to set the dash style and width of line shapes. After selecting an existing line segment or polyline, click on the third button to select a new dash style. Clicking on the fourth button will allow you to set the width of the line.

2.3.5 Font Palette: Vector and System Text Fonts

Clicking on the fifth button (shown as "Ab") allows you to change the font of a selected text primitive. The first eight fonts on the font palette are called vector fonts. These fonts are built into SIMSCRIPT graphics and will look identical regardless of which operating system is being used. Vector fonts can be arbitrarily resized and rotated to any angle. The last button on the font palette allows you to select from one of the fonts loaded with the operating system. Clicking on the "ST" button will display a "font browser" dialog that will allow you to specify a font name, point size and font style.

These system fonts generally look better than vector fonts, but are not guaranteed to be portable. Text primitives having system fonts will remain the same size regardless of parent window geometry.

2.3.6 Color Palette: Change the Color of Anything

The color of any shape can be changed by first selecting it then choosing a color from the color palette. The color palette is shown when you click on the bottom button.

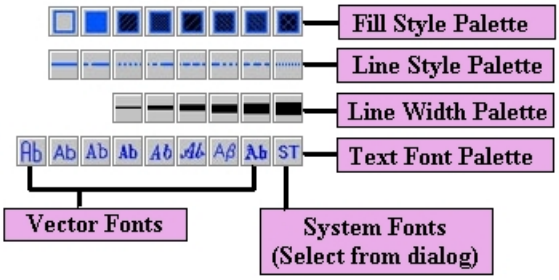

Figure 2-4:Fill, Line and Text Palettes

2.3.7 Selecting, Moving and Resizing Shapes

When you click on a shape in the icon editor it will be surrounded by four green squares called *resize handles*. In addition, the toolbar will reflect the color and style of whichever shape is currently selected. You can select multiple shapes by clicking on the background and dragging a selection rectangle over the shape or shapes you wish to choose. Another way to make multiple selections is to hold down the <Shift> key while clicking on shapes. Use the "Edit / Select All" menu to select the entire icon. Click on the background to de-select all shapes.

The selected shape(s) can be moved by clicking and dragging with the mouse. To resize a selected shape, drag any of the four green resize handles.

The "Cut", "Copy" and "Paste" items from the "Edit" menu can be used on any selected shape or shapes. To delete any shape, select it then press the <Delete> key.

2.3.8 Editing Points

Clicking on a line or polygon that is already selected will permit you to edit the points that define the shape. When in point edit mode, each point along a shape's boundary is marked by a green square called a *point marker*. You can change the shape by dragging any of the point markers with the mouse. New points can be added to the shape by

clicking in between two point markers. You can delete a point marker by clicking on it then pressing the <Delete> key. Click on the background to leave point edit mode.

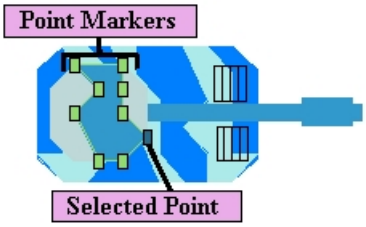

Figure 2-5:Editing Points

2.3.9 Editing Text

You can change the text shown in an icon by double clicking on it showing the "Text property" dialog box. From this dialog you can specify horizontal and vertical alignment, an angle of rotation (in degrees), and the actual text string. The edit box will accept the <Return> key allowing you to create multi-lined text shapes. Horizontal text alignment will apply to each individual line in multi-lined text. However, vertical alignment applies to the bounding box of all lines of text.

2.3.10 Changing Stacking Order

If you wish for a shape to appear on top of all other shapes in the icon, select it then use the "Edit / To Front" menu item. The "Edit / To Back" menu can be used to place the selected shape underneath all the other shapes. The "To Front" and "To Back" buttons on the top toolbar can be used as shortcuts. You can even select multiple shapes to be brought to front or back.

2.3.11 Using the Grid

You can turn on the grid using the "Edit / Grid" menu. Spacing of grid lines can be specified by SETWORLD.R units or by total number of grid-lines. The "Snap to grid" feature is useful for making precise alignment of shapes in the icon. When this feature is turned on, all size, move and point edit operations will result in the modified component being aligned to the closest intersection pair of grid lines.

2.3.12 Changing Properties of the Whole Icon

The "Icon Properties" dialog box can be displayed using the "Edit / Details" menu. This will allow you to specify the following:

Name

This is the name of the icon that is referenced from your application program. This value should be passed to the SIMSCRIPT "SHOW" or "DISPLAY" statement in your code. Usually icon names end with a ".icn" extension.

Priority

The priority can be used to specify the behavior of two overlapping icons in an application program. Icons given a higher priority number will appear on top or lower priority icons. This is the same value used for SEGPTY.A attribute of the icon's display entity in your program.

SETWORLD.R Parameters

The Xlow, Xhigh, Ylow and Yhigh fields contain the dimensions of the world coordinate system. Usually these same values are passed to the SIMSCRIPT "SETWORLD.R" routine. If SETWORLD.R is being used in your program, it is important to set these fields accordingly to ensure that the icon appears to be sized correctly when display in the application.

Center Point

The "X" and "Y" fields under "Center Point" identify the *origin* of your icon. When the application program sets the location of an icon using a "DISPLAY <icon ptr> AT" or "let LOCATION.A(<icon ptr>) =" statement, the icon is positioned relative to this origin. You must clear the "Automatic recenter" check box before you can change the values. If you are designing a static background, and want the objects to appear in your program exactly where they appear in the editor, Set the center point to match the Xlow and Ylow attributes under the **`SETWORLD.R`** parameters.

Automatic Recenter

Usually, if you are created an icon that is to be used as a "dynamic" graphic entity, you will want to mark the geometric center point of an icon as its origin. If this is the case, make sure the Automatic recenter box is checked. If your icon appears in the application program to be positioned incorrectly, try enabling this option

Allow icons to scale with world

This check box defines how the icon is scaled when used in the application program. If this option is checked, the icon will automatically be scaled according to the world coordinate system defined by the application program. If this option is not set, the shape will stay the same size no matter what values are passed to **`SETWORLD.R`**.

Example3: Creating a two wheeled "cart"

Begin by creating a new project and adding a new icon to it. (Right click on "graphics.sg2", select "new". Select "Icon" from the list, type "icon3.icn" into the text box then click on "Create")

a) Turn on the grid by selecting the "Edit/Grid" option. Check "Snap to Grid" and "Show grid" in the dialog box, then click OK.
b) Create a rectangle by clicking on the "Mode Palette" button (shown with an arrow) at the top of the icon editor toolbar. Select the rectangle from the list. Click down and up in two different locations in the canvas to define the rectangle.
c) Drag the rectangle to the middle of the canvas. Resize the rectangle by dragging the small green boxes on its corners.
d) Set rectangle fill-style by clicking on the Style button (second from the top of the icon editor toolbar).
e) With the rectangle selected, click on the bottom button in the toolbar to choose a color.
f) Add a wheel to the cart by clicking on the mode button then choosing the "circle". Click (down then up) on the bottom left corner of the rectangle, then click a couple of grids away to define the size of the circle.
g) With the circle selected, use the "Edit/Copy" menu then the "Edit/Paste" menu to make a copy of the circle. Drag the duplicate to the lower right corner of the rectangle.
h) To resize the whole cart, first drag the mouse over all visible primitives selecting the rectangle and two circles. (Or click on each primitive while holding down the "shift" key.) Use the "Edit/Group" menu to create a single group. Drag one of the four green squares to resize the cart. Use the "Edit/Ungroup" option to allow individual primitives to be positioned.
i) Click on "Edit/Details" to show the "Image Properties" dialog box. Verify that the name of the icon is correct and that the "Automatic Recenter" box is checked. Use the "File/Save" menu to save the icon.

2.3.13 Using JPEG Images in Your Icon

In certain cases, you may want to use an image in your icon that was creating from a different source. For example, you may already have a highly detailed background image that you wish to show in your program. Bitmap or "raster-file" image allow arbitrary detail in the picture and very fast drawing speed in your application. SIMSCRIPT Studio allows you to import JPEG raster file images into your icon. In addition, SIMSCRIPT provides some predefined raster images that can also be added to your icon.

Before importing JPEG files into the icon editor, it is recommended that you first copy these files into the same directory as your "graphics.sg2" file. Use the "Edit / Insert JPEG" menu to display the "Import" dialog. Click on the "Browse Files" button to select a JPEG file name. You can also select one of the built in raster images by clicking on the "Browse Resources" button. The resulting dialog will show samples of all of the

predefined images that can be added to your icon. Click on "OK" once you have either chosen a file or predefined image.

The JPEG image can be moved and resized the same as any other rectangular shape.

To specify options for JPEG images, double-click on the image in the icon editor. The following options are available:

Resizable:
If checked, you can change the width and height of this object in the editor. You can specify the size of the JPEG in World coordinate units. This also means that the image will change in pixel size in your application whenever the user resizes the window. If not checked, the image will always remain its original size (in pixels) in your program regardless of window size or viewing transformation. In this case the image may occupy more or less of the normalized world coordinate space as the user resizes the window.

One drawback of using a resizable JPEG image is that its quality may be reduced as it is resized.

H. Align:
Using the alignment features, your program can position the left, right, top, or bottom edge or any corner of the image without knowing its size in coordinate units. This is especially useful for non-resizable images, where the extent of world coordinate space occupied by the image is not known and varies with the size of the window. Click in the combo box and select from Left, Center, or Right alignment of the image.

V. Align:
Click in the combo box and select from Top, Middle, or Bottom alignment of the image.

Icon Name:
You can change the name of the JPEG file here. Click on the "Browse Files" to locate a jpeg file, or "Browse Resources" to select one of the built in raster images. The icon name must be specified WITHOUT the ".jpg" extension.

Browse Files:
Clicking on this button will allow you to select a new JPEG file for the image. The image file should be located in the same directory as "graphics.sg2".

Browse Resources:
Allows you to select from the set of predefined raster images. 24x24 and 32x32 images are available.

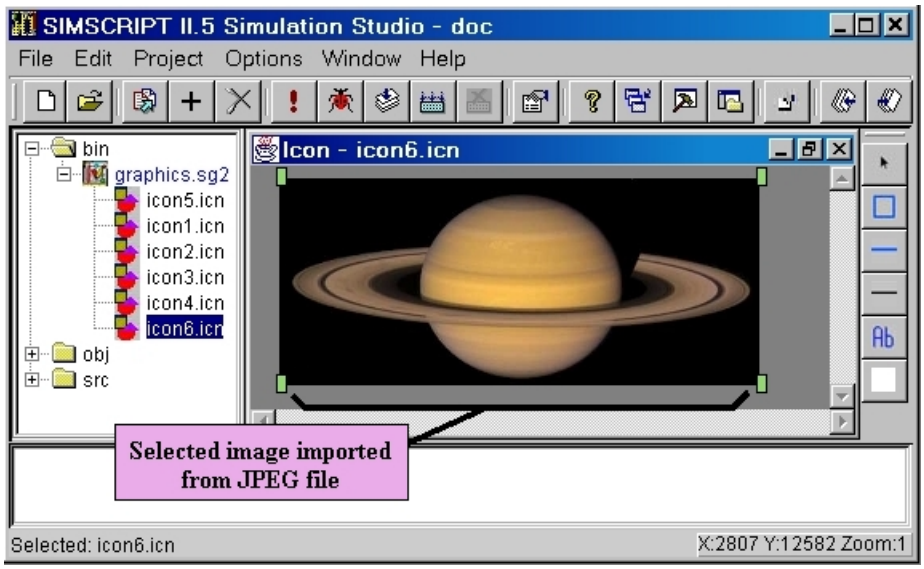

Figure 2-6:JPEG file imported into the Icon Editor

2.3.14 Copying Icons From Other Projects

Users who are maintaining more than one project may need to copy icons from project to project. SIMSCRIPT Studio provides an import capability that you can access by right-clicking on the "graphics.sg2" and selecting "import" from the popup menu.

From the "Import" dialog, select another "graphics.sg2" file and click on the "SG2 Import" button. A list will be displayed allowing you to select which icons you wish to copy into your project. Click while holding down the "Shift" key to select multiple items.

2.4 Declaring Icons as Entities

The principal elementary objects in any SIMSCRIPT simulation program are processes and temporary entities. SIMSCRIPT provides an easy way to associate icons with the entities in your model. SIMSCRIPT supports both `GRAPHIC` entities and `DYNAMIC GRAPHIC` entities. `DYNAMIC GRAPHIC` entities can move across the screen while `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.

2.5 Predefined Attributes of Graphic Entities

When as entity is declared as `GRAPHIC` or `DYNAMIC GRAPHIC,` SIMSCRIPT will automatically add several additional attributes to your entity that help your program to control the icon. The following additional attributes are will be created:

`LOCATION.A`

This attribute allows the program to specify the location of the object with respect to its origin. It must be set to a value produced by the system function **`LOCATION.F (xpos, ypos)`** as follows:

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

`ORIENTATION.A`

Use this attribute to rotate the icon about its origin. The rotation is specified in radians, counterclockwise from 3 o'clock. For example, to rotate your icon 90 degress:

```
Let ORIENTATION.A (entity) = pi / 4.0
```

`SEGID.A`

Returns a segment identifier that can be passed to one of several SIMSCRIPT routines that operate on segments. See Chapter 3 for more information about segments.

2.6 Animating Icons declared as Dynamic Graphic Entities

Dynamic graphic entities have attributes that allow you 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. Use the **`VELOCITY.A ( entity)`** 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 function **`VELOCITY.F (speed, theta)`**, where **`speed`** is a real value in Real World Coordinate Units per Simulated Time Unit, and **`theta`** is the direction of motion in radians.

For example, suppose you want to move a dynamic graphic entity `CART` straight up at the speed of 100 coordinate units per time unit:

```
Let VELOCITY.A(CART) = VELOCITY.F(100, PI.C/2)
```

Pass the real world coordinates of the object's starting position to **`LOCATION.A`**. Suppose you want the object to initially be displayed at coordinate (0, -100):

```
Let LOCATION.A(CART) = LOCATION.F(0.0, -100)
```

If you want the object to continue to be refreshed but not moving, set **VELOCITY.A to VELOCITY.F(0., 0.).** Setting **VELOCITY.A** to 0 will not only stop it from moving across the screen but also remove it from the internal animation queue.

```
Let VELOCITY.A(entity) = VELOCITY.F(0.,0.)  '' Stop motion
Let VELOCITY.A(entity) = 0                  '' Stop animation
```

In addition to the **VELOCITY.A** attribute, display entities have an attribute called **MOTION.A**. This attribute is a sub-program variable that contains the routine called when SIMSCRIPT needs to update the position of an icon that has a velocity. Normally, you will just use the default motion routine which implements linear movement. The motion routine takes the display entity as its only parameter. If you chose to provide your own motion routine, you can use the '**CLOCK.A**' attribute to keep track of the last simulation time.

```
. . .
Let MOTION.A(ICON.PTR) = 'LINEAR.MOTION'
. . .
Routine LINEAR.MOTION(ICON.PTR)
Define ICON.PTR as a pointer variable
Define TDELTA as a real variable

Let TDELTA = TIME.V - CLOCK.A(ICON.PTR)

Let LOCATION.A(ICON.PTR) = LOCATION.F(
      LOCATION.X(ICON.PTR) + TDELTA * VELOCITY.X(ICON.PTR),
      LOCATION.Y(ICON.PTR) + TDELTA * VELOCITY.Y(ICON.PTR))
End
```

2.7 Simulation Time and Real Time

The value of **TIMESCALE.V** establishes a scaling between real-time and simulation time. Setting `TIMESCALE.V = 100` establishes a one-to-one mapping of simulation time units and real elapsed seconds—if the

Real Time (in seconds) = **TIMESCALE.V** * simluation time units / 100

Therefore, 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. There is no guarantee that this ratio of real time to simulation time will be maintained as the simulation runs. When there is not enough processing speed, additional elapsed real time will be taken.

Example 2.4: Moving an icon around in your program
In this example, use the icon created in Example 2.3. When you run the program, the icon will move from the lower left corner (0,0) to the upper right corner of the window.

```
Preamble
Processes include CART
Dynamic graphic entities include CART
End

Process CART
Let LOCATION.A(CART) = LOCATION.F(0.0, 0.0)
Let VELOCITY.A(CART) = VELOCITY.F(4000.0, PI.C / 4.0)
Work 10 units
Let VELOCITY.A(CART) = 0   '' stop the cart
Work 5 units
End

Main
Let TIMESCALE.V = 100      '' 1 second per time unit
Activate a CART now
Show CART with "icon3.icn"
Start Simulation
End
```

2.8 Coordinate Systems

In SIMSCRIPT size and position of icons is not specified in pixel offsets, but in more generalized (and portable) world coordinate units. By default, icons are positioned with respect to what is called "Normalized Device Coordinates". Using this system, the point (0,0) is placed in the lower left hand corner of the canvas, while (32767,32767) is the upper right corner. Given the needs of the application, it may be more convenient to use a different mapping of coordinates.

The SIMSCRIPT **SETWORLD.R** routine is used to specify your coordinate system. But before this function is called, the viewing transformation number must be set to a value between 1 and 15. For example to identify a coordinate system with (-500,-500) in the lower left corner of the window and (500,500) in the upper right corner:

```
Let VXFORM.V = 1
Call SETWORLD.R (-500.0, 500.0, -500.0, 500.0)
```

At the time an icon is made visible, it is sized and positioned with respect to whichever world is currently identified by VXFORM.V. Therefore, whenever you define a custom coordinate system, each icon to be drawn in that world must be told its dimensions in the Icon Editor. While editing each icon, use the "Edit/Details" menu to show the "Icon Properties" box. Enter the SETWORLD.R coordinates into the dialog box.

Example 2.5: Defining your own coordinate system
Suppose we want to show an icon in the world with (0,0) at the lower left corner and (1000,1000) marking the upper right corner. From the icon editor, Create "icon5.icn"

and draw some shapes. Use the "Edit/Details.." option to show the Icon Properties dialog. Enter the values 0, 1000, 0, 1000 into Xlow, Xhigh, Ylow, Yhigh. Also, make sure that both the "Allow icons to scale" and "Automatic recenter" check boxes are checked. The following code should be used:

```
Main
Define ICON.PTR as a pointer variable
Let VXFORM.V = 1
Call SETWORLD.R(0.0, 1000.0, 0.0, 1000.0)
Display ICON.PTR with "icon5.icn" at (500.0, 500.0)
Read as /
End
```

You can implement PAN and ZOOM operations using **SETWORLD.R**. To zoom in, increase the values of XLO and YLO while decreasing XHI and YHI. Pan left or right by adding the same negative or positive constant to XLO and XHI. Pan up or down by adding a negative or positve constant to YLO and YHI.

Example 2.6: Using SETWORLD.R to implement Zoom
To zoom into the above icon, use the same code and icon as in Example 2.5, but specify different parameters to SETWORLD.R. The result will be "bigger looking" shapes.

```
Main
Define ICON.PTR as a pointer variable
Let VXFORM.V = 1
Call SETWORLD.R(250.0, 750.0, 250.0, 750.0)
Display ICON.PTR with "icon5.icn" at (500.0, 500.0)
Read as /
End
```

2.9 Editing Background vs. Movable Icons in the Icon Editor

When creating your icon in the SIMSCRIPT Studio Icon Editor, it is important to bring up the "Icon Properties" dialog to make sure that the attributes correspond to the desired application. In some circumstances the program is responsible for defining the position of the icon at runtime (by setting the **LOCATION.A** attribute or rotating using **ORIENTATION.A**). We will call this category of icon "movable". (Remember that both static and dynamic icons can be repositioned through the **LOCATION.A** attribute.

2.9.1 Editing Movable Icons

When editing the properties for one of these movable icons, you should inspect the center-point fields. This coordinate defines the location (in the icon editor) of the origin or "hot-spot" on the icon that is positioned by the call to **LOCATION.A**. The object will be rotated about this point if the **ORIENTATION.A** attribute is assigned. Usually you will want the origin to be at the geographic center of the icon, in which case you should check the "Automatic recenter" box.

If the "Allow icon to scale with world" check box is cleared, this will prevent the icon from changing size based on changes made via **SETWORLD.R**. You should clear this checkbox if you do not know the dimensions of the world ahead of time, or wish for the icon to remain a fixed size as it appears in the application. The previous example program uses a movable icon with the "Allow icon to scale with world" flag checked.

2.9.2 Editing Background Icons

Now suppose that you are creating a graphical background for your application that will only be displayed once. You usually want the background icon to look the same in your program as it does in the icon editor. Portions of the background may need to be laid out at precise coordinates to match up with other "movable" icons that are to move about the background. To ensure that what you see in the editor matches what you see in your program, double-click on the background of the icon editor window to display the "Icon Properties" dialog. Set the values in this dialog accordingly:

1) Clear the "Automatic recenter" box.
2) Assign the same values for Xlow, Xhigh, Ylow and Yhigh as you are passing to the **SETWORLD.R** routine in your program.
3) Set center point X and Y to match the Xlow and Ylow fields in the "SETWORLD.R Parameters".
4) Check the "Allow icon to scale with world" box.

If these rules are followed you can use the coordinate position indicator in the status bar of the SIMSCRIPT Studio frame window to help you accurately position and size objects in the background. This indicator lists the coordinate point that the mouse is currently hovering over. Using the grid may also be helpful. It is activated using the "Edit/Grid" menu.

2.10 Viewing Transformations

SIMSCRIPT allows more advanced coordinate mapping called the *viewing transformation*. Using these transformations, a single window can show more than one coordinate system, each occupying a separate region of the window. Your program defines the size and position of these regions.

Before defining an individual viewing transformation, the global **VXFORM.V** must be set to a value between 1 and 15 and will be used identify the transformation. The default transformation, **VXFORM.V = 0**, represents the entire NDC space described above and cannot be changed. Set **VXFORM.V** to the id of the view you wish your icon to appear in before that icon is loaded using the **show** or **display** statement. Icons that extend outside the boundaries of the coordinate system space are partially displayed (clipped).

Once VXFORM.V has been set, the program can call **SETWORLD.R** to specify a unique coordinate system for that particular viewing transformation. You can then use the **SETVIEW.R** call to define a viewport within the canvas of the window.

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

Each viewport will have its own coordinate space and occupies a portion of the window space. When calling SETVIEW.R specify this portion of window space using Normalized Device Coordinates (NDC units) where (0,0) is the lower left hand corner of the window and (32767,32767) is the upper right corner. Figure 2.7 shows a typical viewing transformation.

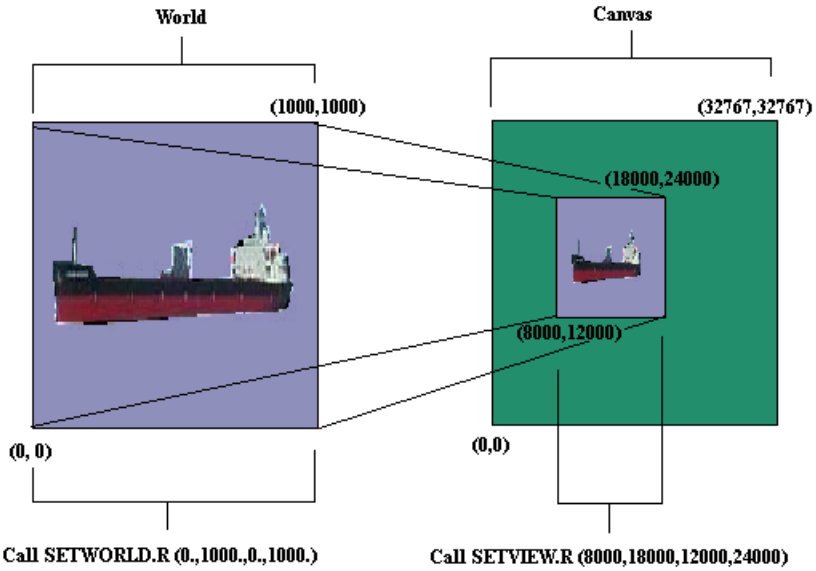

Figure 2-7: The viewing transformation

SETVIEW.R allows for cases where v.xlo > x.xhi or v.ylo > v.yhi. In this case the world coordinate space appears flipped over.

Example 2.7: Multiple views in the same window
In this example we show two worlds in the same window. The first view will occupy the lower left corner of the canvas and be identified by VXFORM.V=1. The second view is in the upper right corner and is referenced by setting VXFORM.V=2.

```
Main
Define ICON1.PTR, ICON2.PTR as a pointer variable

Let VXFORM.V = 1
Call SETWORLD.R(0.0, 1000.0, 0.0, 1000.0)
Call SETVIEW.R(0, 16383, 0, 16383) '' xlo,xhi,ylo,yhi
Display ICON1.PTR with "icon5.icn" at (500.0, 500.0)

Let VXFORM.V = 2
Call SETWORLD.R(250.0, 750.0, 250.0, 750.0)
```

```
Call SETVIEW.R(16383, 32767, 16383, 32767)
Display ICON2.PTR with "icon5.icn" at (500.0, 500.0)

Read as /
End
```

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 you can 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)
```

2.11 Selecting an Icon in Your Program

In certain instances, you may wish to allow the user to click on either dynamic or static icons with the mouse. SIMSCRIPT supports both synchronous and asynchronous selection modes. If you have coded for synchronous selection, your program waits until the user has clicked on an icon in the window. Using asynchronous selection the user can click on an icon while a simulation is running, and a designated block of code will be automatically executed.

2.11.1 Synchronous Selection

The routine **PICKMENU.R** is the easiest way to allow a user to select from a pre-defined list of icons. This routine is defined as follows:

```
Routine PICKMENU.R given ICON.PTR.ARRAY(*) yielding INDEX
```

Each element of 'ICON.PTR.ARRAY' is a graphic entity pointer. This routine waits for the user to make a selection using the mouse. The user must click inside the bounding box of one of the entities to select it. The index of the highest priority selected entity is yielded in the variable 'INDEX'. If the user clicks outside all of the entities, 'INDEX' is set to zero and the routine returns.

2.11.2 Asynchronous selection

In order to allow users to click on an icon while the simulation is running, your program must support asynchronous selection. If your program is already supporting an asynchronous menubar or palette, then you can add the code to handle mouse clicks in the control routine for the form. Whenever a user clicks anywhere in the window, the control routine for the menu-bar or palette is called with the 'field name' parameter set to

the text string "BACKGROUND". Also at this time, the global variable G.4 has been set to the segment id of the selected icon. Determining which icon is selected is a matter of comparing G.4 to the **SEGID.A** attribute of all icons for which selection is meaningful to the program.

Another common way to support this type of selection is by using the routine **READLOC.R** with a *style* value of 16. Using this method, you must create special process in your code whose sole purpose is to handle asynchronous mouse clicks. In a loop, the process will first call **READLOC.R(0,0,16)**, which will return after a mouse click is detected. The program then compares the global variable **G.4** to the **SEGID.A** attribute of any icon that can be selected.

Example 2.8: Select an icon during simulation

For the following example, use "icon3.icn" (cart icon) from Example 2.3. This small program will allow you to click on the cart with the mouse.

```
Preamble
Processes include CART, MOUSE.MONITOR
Dynamic graphic entities include CART
End

Process CART
Let LOCATION.A(CART) = LOCATION.F(0.0, 0.0)
Let VELOCITY.A(CART) = VELOCITY.F(2000.0, PI.C / 4.0)
Work 9 units
End

Process MOUSE.MONITOR
While VELOCITY.X(CART) gt 0.0 do
  Call READLOC.R(0.,0.,16)
  If G.4 eq SEGID.A(CART)
    Write as "Icon was selected!", /
  Else
    Write as "Nothing was selected!", /
  Always
Loop
End

Main
Let TIMESCALE.V = 100          '' 1 second per time unit
Activate a CART now
Activate a MOUSE.MONITOR now
Show CART with "icon3.icn"
Start Simulation
End
```

2.12 Attaching a dynamic text value to your icon

Suppose you wanted to show a text string displaying a status value along with the icon representing some object in a simulation. If the icon is dynamic, you will want this text to move with the icon automatically. Using the SIMSCRIPT Studio icon editor, you can tag a text primitive in your icon as being "Definable", then in your program set the text string by setting the **DTVAL.A** attribute of the **ICON.A** attribute of your graphic entity.

```
Let DTVAL.A(ICON.A(ICON.PTR)) = "Hello There"
```

After setting this attribute, remember to redisplay the icon to enable the user to see the changes.

Example 2.9: Dynamically label an icon

For this example, create a new icon named "icon9.icn" similar to the one created in Example 2.3. Add a text primitive to the icon by clicking on the top button in the icon editor's toolbar, then selecting the "text" button. After you have placed the text in the icon, click on the "Properties" button. Set the "Definable using DTVAL.A" check box, exit the dialog box, and then save the icon. The following code will increment a counter attached to the cart every second.

```
Preamble
Processes include CART
Dynamic graphic entities include CART
End

Process CART
Let LOCATION.A(CART) = LOCATION.F(0.0, 0.0)
Let VELOCITY.A(CART) = VELOCITY.F(2000.0, PI.C / 4.0)
For I = 1 to 9 do
  Let DTVAL.A(ICON.A(CART)) = ITOT.F(I)
  Display CART
  Work 1 unit
Loop
End

Main
Let TIMESCALE.V = 100          '' 1 second per time unit
Activate a CART now
Show CART with "icon3.icn"
Start Simulation
End
```

2.13 Changing the Color of an Icon

As your simulation is running, it may be useful to change the color of a portion of an icon to, for example, indicate a state change. From within the SIMSCRIPT Studio icon editor, you can mark primitives in your icon as being "definable by **DCOLOR.A**". Marked components will be re-colored by SIMSCRIPT after you set the **DCOLOR.A** attribute of the icon's **ICON.A** attribute pointer value.

From the icon editor, select any one of the primitives then click on the "properties" button on the top toolbar. From the dialog box, check the '**Define color using DCOLOR.A'** check box, and save the icon. In your program, use the following code to change the color of the icon:

```
let DCOLOR.A(ICON.A(ICON.PTR)) = COLOR.INDEX.VALUE
display ICON.PTR
```

2.14 Destroying and Erasing Icons

The image of a graphic entity is placed on the display surface when the first DISPLAY statement is executed. You can simply erase the icon from the window without destroying the display entity by using the ERASE command:

```
Erase ICON.PTR
```

Execution of a DESTROY statement will destroy the entity, free all memory used by graphics, and erase the icon.

```
Destroy ICON
```

Or

```
Destroy this ICON called ICON.PTR
```

3. Segments

Normally, it is recommended that programmers use the SIMSCRIPT Studio Icon Editor to create and save icons using the user interface. However, there are some circumstances where icons must be created by the program. Perhaps the icon is created based on one or more data values that are known only when the program is run. The SIMSCRIPT run-time library provides a rich set of routines that can be used to build your own icon.

To draw static graphical images or graphics that are not linked to a display entity, SIMSCRIPT provides "segments". A segment is basically a grouped collection of one or more lines, fill areas, graphic text etc. There are several routines that let you build, manipulate, and destroy segments.

If you want to use a dynamic or static display entity whose image is defined at execution time, a "display routine" can be specified for the entity. The display routine will be called whenever SIMSCRIPT needs to draw your icon. The display related statements and attributes like **VELOCITY.A** and **LOCATION.A** that were explained in Chapter 2 will still work if the program is using a custom display routine.

3.1 Color

In SIMSCRIPT the color of any primitive can be 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 to be "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** (see chapter 9). For example, to set a window's background color to "blue":

```
call GCOLOR.R( 0, 0, 0, 1000 )
```

The first sixteen color values can be defined in a file called *colors.cfg*. This file will be read in automatically (if it exists) before program starts executing, and should therefore be kept in the same directory as the executable. Each line in this file defines a color index as follows: <Index> <red> <green> <blue>. Here are some sample entries:

```
0      0      0      0
1   1000   1000   1000
2      0      0    500
3    250    250   1000
...
```

```
15  1000    250   1000
```

3.2 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 a "segment." Each segment has an identifier and comprises a logical grouping of related graphic primitives. SIMSCRIPT 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 is identified by your program using its “segment id”. 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**. After this routine is called, you can make calls to routines that draw graphic primitives like FILLAREA.R and CIRCLE.R.

CALL CLOSE.SEG.R
Closes the currently open segment and makes it visible. **SEGID.V** is set to zero.

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. (Segment priority is explained later).

CALL GVISIBLE.R (segid , vis)
Make a segment visible or invisible, where **vis** is an integer; 0 = invisible, 1 = visible.

CALL GDETECT.R (segid , sel)
Make a segment selectable with the mouse where **det** is an integer; 1 = selectable, 0 = not selectable.

CALL GHLIGHT.R (segid , hi)
Where **hi** is an integer; 0 = off, 1 = highlight on. Sets the highlighting status of a segment. When highlighted, the entire segment is drawn using color index number 15.

3.3 Adding Primitives to a Segment or Display Routine

After the call to **OPEN.SEG.R** (or anywhere in a display routine) your program can call routines to add the polygons, polylines, circles, text to the segment.

Many of the routines below accept a set of some point values to define the shape of the primitive. These points are specified in SIMSCRIPT as arrays of pairs of coordinate values that live in the Cartesian space whose dimensions are given through the SETWORLD.R routine described in Chapter 2.

The first subscript in a coordinate pair selects either x-coordinates (index = 1) or y-coordinates (index = 2); the second subscript determines a point. The coordinate arrays are stored in REAL (not DOUBLE) mode. For example, to specify points at (40,-100) and at (40,0) use the following 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.
```

3.4 Drawing Filled 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(*))
Adds a polygon to the segment given its vertices. The last point is automatically joined to the first point. The present fillstyle and fillcolor are used.

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 a 'filled' arc (like a pie slice), 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.

3.5 Drawing Lines

The primitive operations in this section generate solid and dotted lines. The graphic style routines should be called first, to set the appearance of the line.

CALL LINESTYLE.R(style)
SIMSCRIPT 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 above).

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(*).

3.6 Drawing Markers

SIMSCRIPT 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)
Sets the color of all markers.

CALL MARKSIZE.R (size)
Sets the size of each marker, in NDC units.

CALL POLYMARK.R (n, points(*))
Writes n markers using the current marker type, color, and height.

3.7 Drawing Text

Text can be written directly onto the window. It is displayed using the text size, font and color attributes. Text can be rotated, and also justified left, right, top, bottom or centered.

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 TEXTCOLOR.R (color)
Set color index to use for drawing text.

CALL TEXTSIZE.R (size)
Sets the height of vector (non-system) text given in NDC units (0-32767). Default text size is 560 units high.

CALL TEXTALIGN.R (horiz, vert)
Set text alignment. Text will be aligned with respect to the (X,Y) coordinate specified in WGTEXT.R. Horizontal alignment values are as follows:

1—Left justified
2—Centered
3—Right justified

Vertical alignment values are as follows:

1—Top of cell
2—Top of character
3—Middle of cell
4—Bottom of cell
5—Bottom of character

CALL TEXTANGLE.R (degrees*10)
Set text rotation angles in tenths of a degree.

CALL TEXTFONT.R (font)
With regard to text fonts, there are two varieties. SIMSCRIPT provides six built in "vector" drawn fonts. Vector fonts can be fully scaled and rotated, and look identical under any operating system. To use one of the vector fonts call this function with one of the following values:

0—basic font
1—Simple script
2—Roman
3—Bold Roman
4—Italic
5—Script
6—Greek
7—Gothic

CALL TEXTSYSFONT.R (family.name, point.size, italic, bold)
The disadvantage of vector fonts, is that the may not look nice when scaled up in size. Drawing speed may also be a concern if lots of text is used in the window. "System" text fonts (or raster based fonts) are acquired from the operating system under which the program is currently running. This call lets you specify the common name of the font, as well as its point size, and whether or not text should be bold or italic. For example, to draw "Hello world" in Times Roman, size 12 italic font, you could use the following code:

```
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 (usually only two degrees are provided.). Calling **TEXTFONT.R** will re-enable vector fonts.

Example 3.1: Drawing some graphics from program code only

```
Main
Define FILL.POINTS, LINE.POINTS as 2-dim real array

'' define a new segment to draw graphics
Call OPEN.SEG.R

'' Draw a filled purple triangle into the segment
Reserve FILL.POINTS(*) as 2 by 3
Let FILL.POINTS(1,1) = 2000    Let FILL.POINTS(2,1) = 30000
Let FILL.POINTS(1,2) = 30000   Let FILL.POINTS(2,2) = 30000
Let FILL.POINTS(1,3) = 16000   Let FILL.POINTS(2,3) = 2000
Call GCOLOR.R(1, 500, 0, 500)  '' purple
Call FILLCOLOR.R(1)
Call FILLAREA.R(3, FILL.POINTS(*,*))

'' Draw green markers on the endpoints of the filled triangle
Call GCOLOR.R(2, 0, 1000, 0)    '' green
Call MARKTYPE.R(3)
Call MARKCOLOR.R(2)
Call MARKSIZE.R(600)
Call POLYMARK.R(3, FILL.POINTS(*,*))

'' draw a single line into this segment
Reserve LINE.POINTS(*) as 2 by 2
Let LINE.POINTS(1,1) = 8000   Let LINE.POINTS(2,1) = 18000
Let LINE.POINTS(1,2) = 24000  Let LINE.POINTS(2,2) = 18000
Call GCOLOR.R(3, 1000, 600, 0) '' orange
Call LINESTYLE.R(2)
Call LINEWIDTH.R(700)
Call LINECOLOR.R(3)
Call POLYLINE.R(2, LINE.POINTS(*,*))

'' draw some text
Call GCOLOR.R(4, 1000, 0, 0)    '' red
Call TEXTFONT.R(2)
Call TEXTALIGN.R(1,2)    '' centered text
Call TEXTSIZE.R(2048)    '' 1/16 of window
Call TEXTCOLOR.R(4)
Call WGTEXT.R("Hello, World!", 16000, 20000)

'' done with segment.  Now display it
Call CLOSE.SEG.R

'' Wait for the user to close the window
While 1=1 do
    Call HANDLE.EVENTS.R(1)
Loop
```

```
End
```

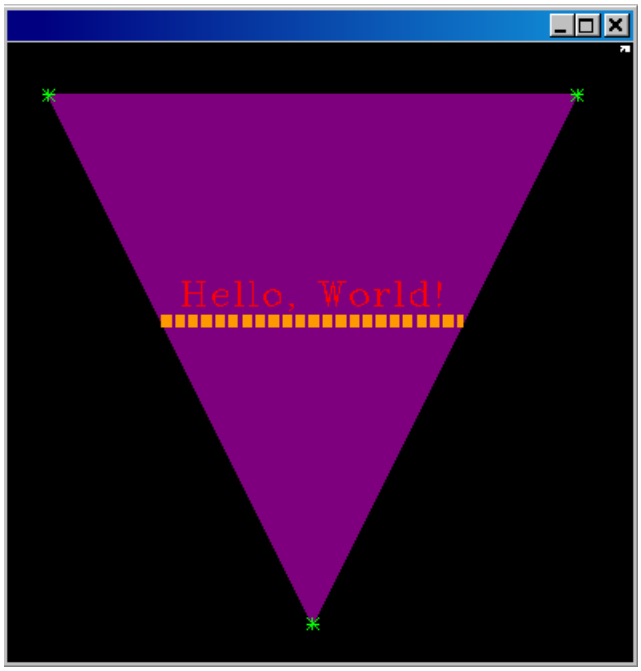

Figure 3-1:Graphics produced by Example 3.1 code

3.7.1 System Font Browser

You may want to allow a user to select one of the available system fonts from a predefined font browser dialog box. This is done by calling the **`FONTBOX.R`** routine

```
call FONTBOX.R given TITLE yielding
      FAMILY.NAME, POINT.SIZE, ITALIC.DEGREE, BOLDFACE.DEGREE
```

The `FAMILY.NAME, POINT.SIZE, ITALIC.DEGREE and BOLDFACE.DEGREE` parameters are identical to those defined by the `TEXTSYSFONT.R` routine. The layout of the dialog box is dependant on the toolkit and operating system as well as the choice of fonts. If the Cancel button is pressed, the "FAMILY.NAME" parameter will be set to "".

Example 3.2: Allowing the User to Select a System Font

This program will immediately show a font browser dialog allowing a system text font to be specified. The selected font's name is displayed in the window

```
Main
Define TITLE, FAMILY.NAME as text variables
Define POINT.SIZE, ITALIC.DEGREE, BOLDFACE.DEGREE as integer variables
Let TITLE = "Select a font"
Call FONTBOX.R given TITLE yielding
```

```
      FAMILY.NAME, POINT.SIZE, ITALIC.DEGREE, BOLDFACE.DEGREE
If (FAMILY.NAME ne "")
    Call OPEN.SEG.R
    Call TEXTSYSFONT.R given
        FAMILY.NAME, POINT.SIZE, ITALIC.DEGREE, BOLDFACE.DEGREE
    Call WGTEXT.R(FAMILY.NAME, 1000., 16000.)
    Call CLOSE.SEG.R
    Read as /
Always
End
```

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 **`FAMILY.NAME`** is set to the empty string.

3.8 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 with a priority value of 'zero' 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.

3.8.1 Using Priority Zero

Segment with priority zero are not redrawn when their bounding box is overlapped by moving objects. A segment with priority zero will be eaten if any other graphical image is drawn on top or underneath.

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.

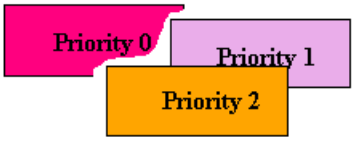

Figure 3-2:Overlapping objects of different priority

3.9 Display Routines

Previously, we have covered cases where an icon is to move around the window over time. However, what if the structure of the icon itself should be dynamic? That is to say, the icon's exact appearance is either not known ahead of time, or changes dynamically. A display routine is an attribute of a graphic entity that can be set programmatically. This routine is called automatically by SIMSCRIPT whenever it is necessary to display an icon (i.e. the program executes a **DISPLAY** or **LOCATION.A** statement). The routine can contain code to either draw the icon from scratch using calls to create output primitives, or to make modifications to an icon created using the icon editor. 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
```

You should not call **OPEN.SEG.R** or **CLOSE.SEG.R** from inside a display routine. Upon entering the display routine, SIMSCRIPT will automatically create a new segment (whose identifier is stored in the global **SEGID.V**). From inside the display routine, call drawing routines like **POLYLINE.R**, **FILLAREA.R**, and **WGTEXT.R**.

Example 3.3: Using an icon whose shape changes over time

In the following case, we show how the use of a display routine can enable your program to draw an icon. In this example the icon editor is not used; all drawing is done entirely within the program code. Here, the display routine draws a triangle using the **ORIENTATION.A** attribute to compute the location of one of its vertices. Note that the TRIANGLE process draws the icon using the **DISPLAY** statement knowing that SIMSCRIPT will automatically call the display routine specified in 'main' to render the icon.

```
Preamble
Processes
  Every TRIANGLE has an ATTR
Define ATTR as a real variable
Graphic entities include TRIANGLE
End
```

```
Process TRIANGLE
Define I as an integer variable
Let DRTN.A(TRIANGLE) = 'V.TRIANGLE'    ''  set display routine
For I = 1 to 36*5 do
  Let ATTR(TRIANGLE) = (I * 10.) * pi.c / 180.0
  Display TRIANGLE
  Work 0.1 units
Loop
End

Display Routine TRIANGLE given TRIANGLE
Define TRIANGLE as a pointer variable
Define PTS as a 2-dim real array
Reserve PTS(*) as 2 by 3     '' 3 points
Let PTS(1,1) = 10000.0       Let PTS(2,1) = 16383.0
Let PTS(1,2) = 22767.0       Let PTS(2,2) = 16383.0
Let PTS(1,3) = 5000.0 * COS.F(ATTR(TRIANGLE)) + 16383.0
Let PTS(2,3) = 5000.0 * SIN.F(ATTR(TRIANGLE)) + 16383.0
Call FILLCOLOR.R(180)
Call FILLAREA.R(3, PTS(*))
Release PTS(*)
End

Main
Let TIMESCALE.V = 100          '' 1 second per time unit
Activate a TRIANGLE now
Start Simulation
End
```

Example 3.4: A moving icon defined by the program

This next example is a more complete graphical simulation that uses a display routine, and a custom defined world coordinate system.

```
Preamble ''Case Study "DYNSHAPE"
'' This shows a simple dynamic graphics output using SIMSCRIPT.
'' It draws a shape and moves it around the screen.
'' This version does not use the Icon Editor and
'' It shows the details for generating an icon by program code only.
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 SIMSCRIPT GRAPHICS indices from numbers to words
Define .RED to mean 1
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
'' Define index for red to be 1
call GCOLOR.R(.RED, 1000, 0, 0)
'' 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'
'' Move diagonally up the window for 10 seconds
Let VELOCITY.A(SHAPE) = VELOCITY.F(200.0, PI.C/4)
Let LOCATION.A(SHAPE) = LOCATION.F(0.0, 0.0)
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
'' Make the shape rotate
Let VELOCITY.A(SHAPE) = VELOCITY.F(200.0, 0.8 * PI.C)
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
```

3.10 Customizing an Icon Defined in SIMSCRIPT Studio

Supposed you want to use SIMSCRIPT Studio's icon editor to create your icon but still need some portion of the icon to be defined at runtime. SIMSCRIPT allows you to specify a display routine for an icon even if it had been loaded from the icon editor. From the display routine you can call routines (like FILLAREA.R) to draw the "run time" portion of the icon, then instruct SIMSCRIPT to display the rest of the icon.

Basically, the **ICON.A** attribute points another display entity containing the portion of your icon defined in SIMSCRIPT Studio. Use the following code inside of your display routine:

```
Display ICON.A(MY.ENTITY.PTR)
```

Be sure in the initialization code that you use the "SHOW" statement to load in the definition of the icon BEFORE setting the DRTN.A attribute to your display routine.

Example 3.5: An icon defined by SIMSCRIPT Studio and program code

For this example, create in SIMSCRIPT Studio an icon called "icon.icn" containing a few shapes. (Use the "Icon Properties" dialog to make sure the icon is centered). When the program is run, your icon will be annotated with text indicating its current position.

```
Preamble
Processes include MOVER
Graphic entities include OBJECT
End

Process MOVER
Define A as a real variable
show OBJECT with "icon.icn"
Let DRTN.A(OBJECT) = 'V.OBJECT'    ''  set display routine
For A = 0.0 to pi.c by 0.01 do
  Display OBJECT at (16000. + 12000. * COS.F(A), 27000. * SIN.F(A))
  Work 0.1 units
Loop
End

Display routine OBJECT given OBJECT
Define OBJECT as a pointer variable
Display ICON.A(OBJECT)
Call WGTEXT.R(CONCAT.F("X: ", ITOT.F(LOCATION.X(OBJECT))), -
4000.,3000.)
Call WGTEXT.R(CONCAT.F("Y: ", ITOT.F(LOCATION.Y(OBJECT))), 1500.,3000.)
End
```

```
Main
Let TIMESCALE.V = 100          '' 1 second per time unit
Activate a MOVER now
Start Simulation
End
```

3.11 Modeling Transformations

In SIMSCRIPT, there is a single modeling transformation that consists of translation, rotation, and scale parameters. Parameters of this modeling transformation can be set by the programmer to produce changes in how all segments and custom icons are drawn.

The modeling transformation is supported by the library routines:

MXRESET.R (entity) Set modeling transform using entity attributes.
MSCALE.R (factor) Scale by factor.
MZROTATE.R (radians) Rotate counterclockwise around origin.
MXLATE.R (xval, yval) Translate (move) in X and Y.

You can call **MZROTATE.R, MXLATE.R, or MSCALE.R** before calling routines to create primitives to cause those graphics to be rotated, translated or scaled. Rotation and scale are performed before translation. The effects of successive calls on both **MZROTATE.R** and **MXLATE.R** are cumulative. Translation should be specified in real world coordinate units.

Call **MXRESET.R(0)** to reset the modeling transformation to zero translation, zero rotation and a scale factor of 1.0. You can pass a display entity pointer to **MXRESET.R** to copy the **LOCATION.A**, and **ORIENTATION.A** attributes of the entity into the modeling transformation.

Keep in mind that whenever your program executes a "DISPLAY *entity*" or "Let **LOCATION.A** *entity* " statements, SIMSCRIPT will automatically call **MXRESET.R** given a pointer to the entity that is being displayed. In this case the current modeling transformation will be lost. If you plan on using a display routine, use of the modeling transformation should be reserved for inside of the display routine. Calls to **OPEN.SEG.R** do not change the modeling transformation.

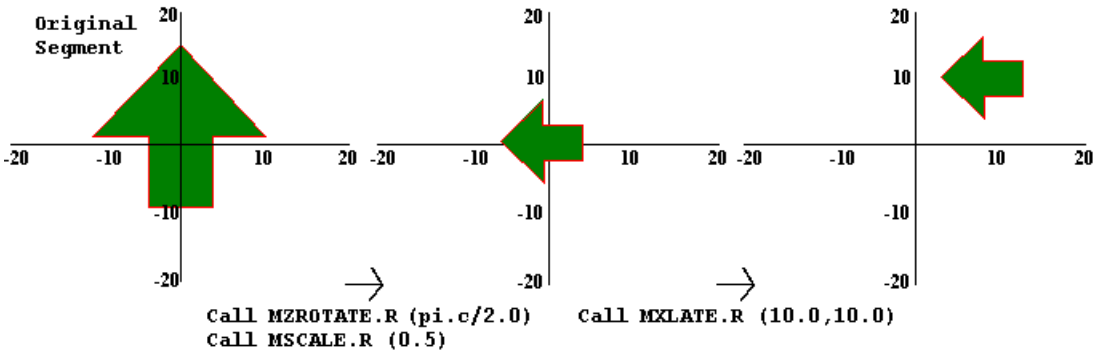

Figure 3-3: Modeling transformations in action

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 a variety of graphs. These graphs are constructed in SIMSCRIPT Studio, and loaded into you program via a **SHOW** or **DISPLAY** statement. Graph types include:

- Histograms,
- Grouped histograms,
- Dynamic bar charts,
- Pie charts,
- X-Y plots,
- Trace plots exhibiting variables traced over time,
- Meters that show a single value

These features are supported by SIMSCRIPT II.5 language enhancements. In general, data is collected with versions of the **TALLY** and **ACCUMULATE** statements. Data is 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 graph—for example, a scalar value can be shown on a dial or level meter, but a 2-d chart, or piechart is required to represent an array of values.

To create presentation graphics:
1. Create a chart, or meter in SIMSCRIPT Studio and add to **graphics.sg2**.
2. Declare the relevant globally-defined variables as DISPLAY in the Preamble.
3. Add statements to the executable code to associate display variables with the graph stored in **graphics.sg2**.

Each of these steps is described below.

4.1 Using SIMSCRIPT Studio to Create and Edit a Graph

SIMSCRIPT Studio Provides a Graph Editor that can be used to create and modify your 2-d charts, pie-charts, clocks, and meters. To get started with the graph editor, click with the right mouse button on the "graphics.sg2" element, then select "new" from the popup menu. In the dialog box, select from the list labeled "Type" the variety of clock, chart, meter or display you with to create. Also provide a name for your graph. This same name will be used in your program code to load the graph (via a **SHOW** or **DISPLAY**

statement). Clicking on the "Create" button will create a new graph editor window in SIMSCRIPT Studio's edit pane.

When a graph editor window is active a **Style** palette becomes attached to the right side of the SIMSCRIPT Studio window frame. This palette allows you to zoom in and out of the graph editor window, and also change the fill style, line style, text font and color of any component on the graph.

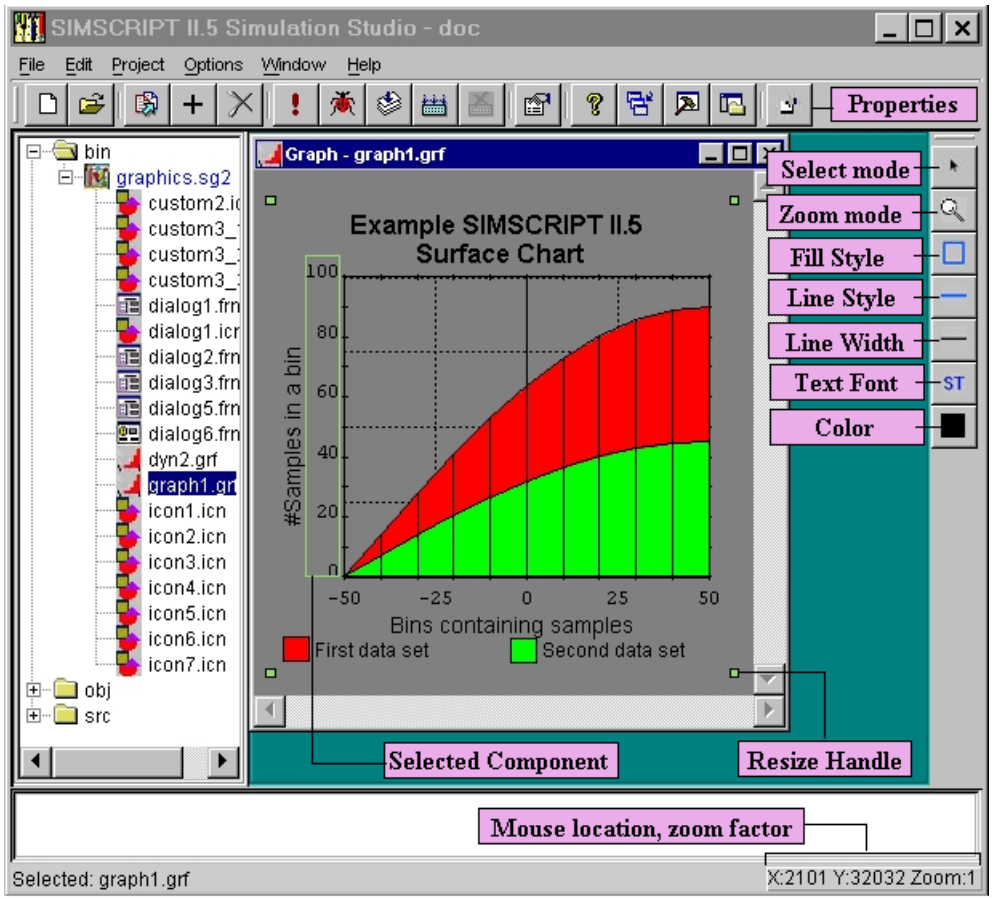

Figure 4-1: The SIMSCRIPT Studio Graph Editor

4.1.1Changing Size and Position of a Graph

There are two modes of operation in the graph editor called *select* and *zoom*. In select mode (default) you can click on various parts of a graph, then change the style or color. You can also move the graph by dragging it with the mouse. At each corner of the graph, you will see a small green box called a *resize handle*. You can drag any of these handles with the mouse to resize the graph. The graph will be displayed in the same size and location in your program as shown in the editor window.

4.1.2 Zoom In and Out

Clicking on the second button from the top of the style palette (magnifying glass) allows you to enter a zoom mode. When in *zoom* mode, clicking on the graph with the *left* mouse button will zoom in, while the *right* button allows you to zoom out. Click on the top button (arrow) to return to *select* mode.

4.1.3 Changing Color, Font, Fill, and Line Styles

The **Style** palette also has buttons that enable you to change fill style, dash style, line width, and text font. Clicking on the bottom button will allow you to make a color change. You can change the style or color of any part of the graph by first selecting the component, then choosing a new style or color from the **Style** Palette.

4.1.4 Changing Data Related Properties

As the graph editor starts up, a graph with default settings will be shown in the editor's window. To change the axis scaling, data-set properties, titles or the name, double click on the graph while in select mode. A "property" dialog box will appear allowing you to change attributes of the graph. It is important to set the *name* field to the same value that is used in your program code to load in the graph (via the DISPLAY or SHOW statement). Graph names usually have the ".grf" extension. See below for a more detailed description of the various graph types.

Table 4-1: Mapping graph types to variable types.

Created in SIMSCRIPT Studio	Variable type	Declaration statement
Chart	Real, Integer, Array	DISPLAY,TALLY, ACCUMULATE
Analog Clock, Digital Clock	Real, Integer, TIME.V	DISPLAY
Dial, Digital Display, Level Meter	Real, Integer	DISPLAY
Pie-chart	Array	DISPLAY
Text Meter	Text	DISPLAY

4.2 Displaying Single Variables in a Meter

A SIMSCRIPT program may have important scalar numerical values that are either global variables or entity attributes that change over time. In SIMSCRIPT you can associate what we will call a *display variable* with one of the graphs saved in your graphics.sg2 file. When the variable changes in your program, the graph is updated automatically to reflect its new value.

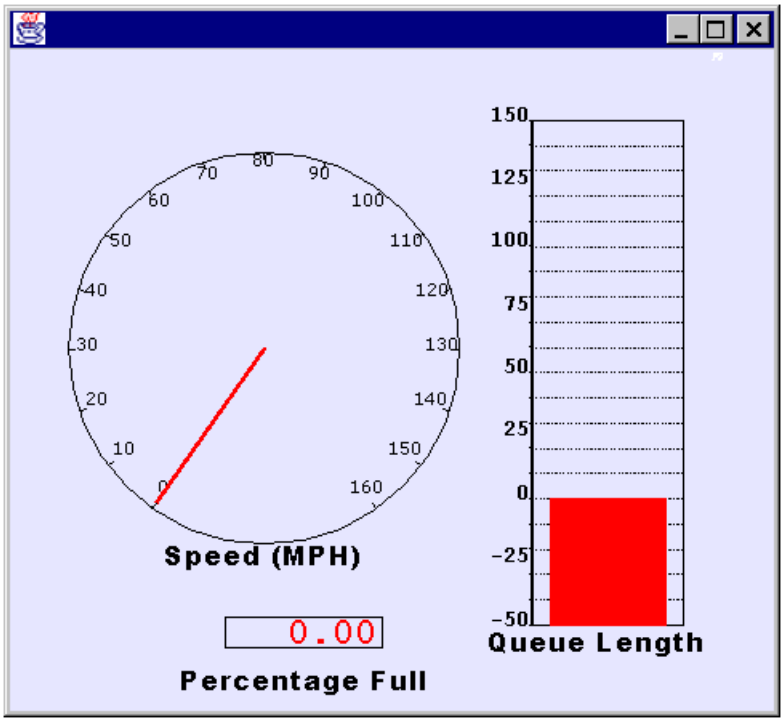

Figure 4-2: Meters in SIMSCRIPT

4.2.1 Creating a Meter in SIMSCRIPT Studio

There are three types of meters that can be created in SIMSCRIPT Studio to represent a display variable—Level meters, dials, and digital displays. (These items can be picked from the "Create new graphic" dialog box described above). From the graph editor, double-click on the meter to display its "Properties" dialog box. This dialog will contain the following:

- **Name** – The name of the graph. Used by your program in a "show" or "display" statement.
- **Title** – Text of title displayed on bottom.
- **Minimum, Maximum** – Defines the range of values shown by the meter.
- **Interval** – Distance between tic marks on the single axis or face (specified in "Axis" coordinates)..
- **Num Interval** – Distance between numbers on axis or face (specified in "Axis" coordinates).
- **Min Theta** – (Dial only) Angle in degrees where the minimum value is placed around the dial circumference. Angle is measured counter-clockwise from 3:00 position.
- **Max Theta** – (Dial only) Angle in degrees where the maximum value is placed around the dial circumference. Angle is measured counter-clockwise from 3:00 position.
- **Field Width** – (Digital display only) Number of places allotted for the entire value (including decimal point).

- **Precision** – (Digital display only) 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 dial.
- **Show Border** – A square background can be shown under the meter face and title.

4.2.2 Monitoring a Single Variable in your Program

In order to add graphical monitoring to a quantity in your program, the quantity must be defined as a display variable. Display variables are defined in the PREAMBLE using the following syntax:

```
DISPLAY VARIABLES INCLUDE variable1, variable2....
```

This declaration is made *in addition* to normal variable declarations. In other words the variables used in this statement must be known to SIMSCRIPT at the time the declaration is made.

The graph to show a single value is created in SIMSCRIPT Studio (see Chapter 4.1). Table 4.1 will show you of the graph types can be created to display the value. At initialization time, your program will have to load the graph's description file called "graphics.sg2". This is accomplished with the SIMSCRIPT "SHOW" or :"DISPLAY" statement.

```
SHOW variable WITH "graph_name"
```

or

```
DISPLAY name1,name2,.. WITH "graph_name"
```

Using the DISPLAY statement will load the graph from "graphics.sg2" and cause the graph to become visible immediately. The SHOW statement will only load the graph. The display or show must be called before the first assignment is made to the monitored variable.

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

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

Example 4.1: Show a single global variable changing over time

Create the graph in SIMSCRIPT Studio by right-clicking on "graphics.sg2" and selecting "new". From the dialog box choose "Level Meter" from the list and set the name to "graph1.grf". Use the following source code:

```
Preamble
Define K as a real variable
```

```
Display variables include K
End

Main
Display K with "graph1.grf"
For K = 0.0 to 100.0 by 0.01
Do
Loop
Read as /
End
```

4.3 Charts

As far as SIMSCRIPT Studio is concerned, Histograms (dynamic and static), time trace plots, and X-Y plots are shown using a single type of graph—the 2-D Chart. A chart can contain one or more *datasets*, each representing some statistic compiled on a global variable in your program. Each dataset can contain a fixed number of "cells" (bar-graph, histogram, surface chart representation) or can collect a new data point each time the monitored value changes (continuous representation). A chart can have a second Y-axis (usually on the right). You can therefore not only show more than one histogram or X-Y plot on the same graph, but show each quantity in a different scale.

4.3.1 Editing a Chart in SIMSCRIPT Studio

For applications requiring a histogram, time trace plot, or X-Y plot, you should create in SIMSCRIPT Studio a "Chart". Create by right clicking on the "graphics.sg2" tag, selecting "new", then picking "Chart" from the list-box. Once in the graph editor, charts behave like the other graphs. You can click on various components (numbering, bars, axis) then use the style and color buttons on the right palette to change the appearance. Double click on the graph to show the properties.

4.3.2 Chart Properties Dialog Box

The main detail dialog box for the chart has buttons for changing the Axes, adding, removing and editing data-sets, as well as specifying labels.

- **Name** – The name used to load the chart into your application program.
- **Title** – The title shown on the top of the chart. The title can have multiple lines of text.
- **Axes on Edges** – If checked, numbering and tic marks will appear on all 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 properties of a data set, select its current name in the list box and then press the **Edit** button.
- **Handling of Multiple Data Sets** –

1. 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.
2. "Side by side" means to show thinner bars next to each other in the same cell. Unlike "stacked" the top of each bar reflects its current value.
3. If the "Overlap" radio button is marked, higher numbered data sets will obscure the lower numbered sets.

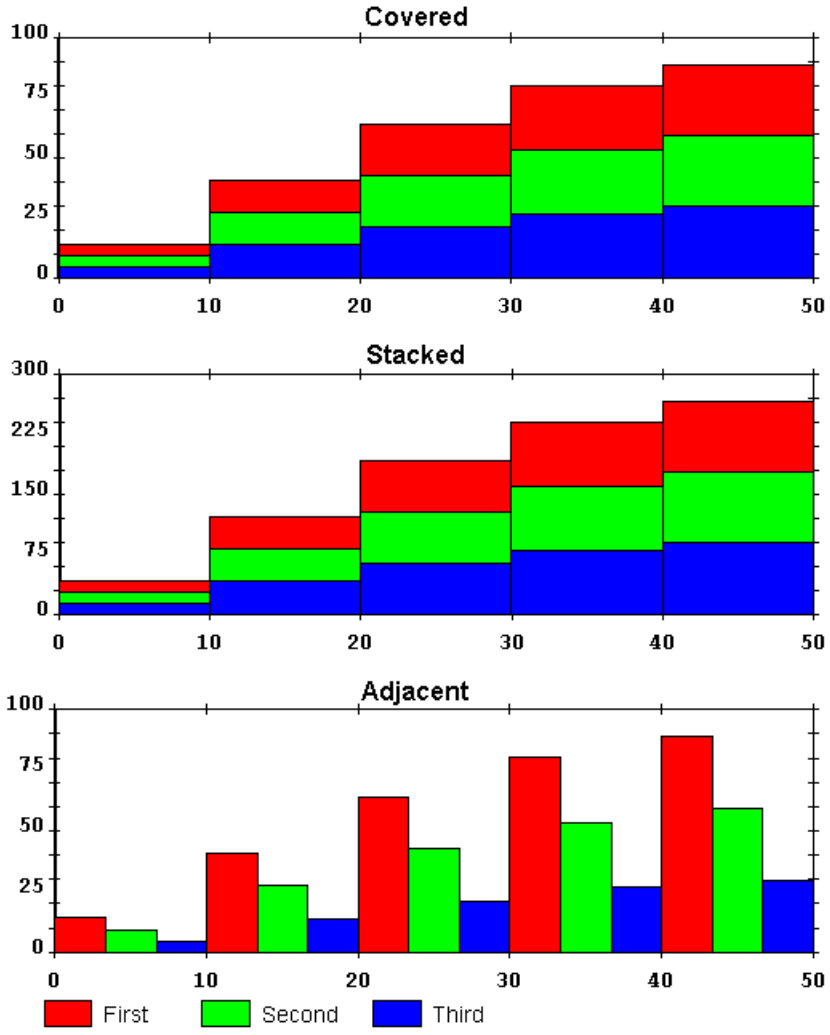

Figure 4-3: Three ways to show multiple data sets together.

4.3.3 X,Y,Y2 Axis Detail Dialog Box

To change the range, numbering interval, or any other property associated with the X axis, click on one of either "X axis", "Y axis" or "Y2 axis" buttons from the "Chart Properties" dialog box. Another dialog will be brought up which contains the following:

- **Title** – Label for axis displayed below or to the left of numbering. The title can contain multiple lines of text.
- **Rescaleable** – Specifies whether the axis will be re-numbered (scaled) when one of the data points extends beyond its limit. In the case or the X-axis, 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 axis will be discarded.
- **Show grid lines** – If this item is on, *grid lines* will be shown crossing the axis.
- **Tics centered, Tics inside, Tics outside** – Defines the tic mark alignment with respect to the axis line. Tics marks can be attached to the axis from their center, left or right sides.
- **Compress data** – (X-Axis only) 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 lo and high values on the axis.
- **Tic Interval (Major & Minor)** – Defines the distance along the axis between consecutive tic marks. If an interval is zero, tic marks will not be displayed.
- **Numbering Interval** – Defines the distance along the axis between consecutive number labels on the axis.
- **Grid line Interval** – Defines the distance along the axis between consecutive grid lines.
- **X,Y, Y2 Intersection Point** – Defines the point (in axis coordinates) along the axis where the perpendicular axis crosses.
- **Data Scaling Factor** – Defines the factor multiplied to that component of all data plotted to the chart at runtime.

4.3.4 Attributes of a Data Set

You can edit individual attributes of a data set from the chart detail dialog box. Select the name of the database you wish to change, and click on the "Dataset" button. This will show the "Dataset Detail" dialog box which contains the following fields:

• **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. Use this type of data set for trace plots, but not for histograms.

• **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 changes by clicking on a bar and selecting a new style from the right palette.
2. **Line** – Plot data cell using a polyline. Clicking on the line from the graph editor window, then selecting a new line type from the right hand palette can change line width and dash style.
3. **Mark** – Use a small marker to represent the data point. Markers should only be used with the "continuous surface" representation.

• **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 for static graphs. In this case, a single polygon (or polyline) will be used to display all cells in the data set.

• **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.

• **Polymark Attributes** – Select from one of six varieties of mark style. You must check the "Mark" box before markers will be used in your graph.

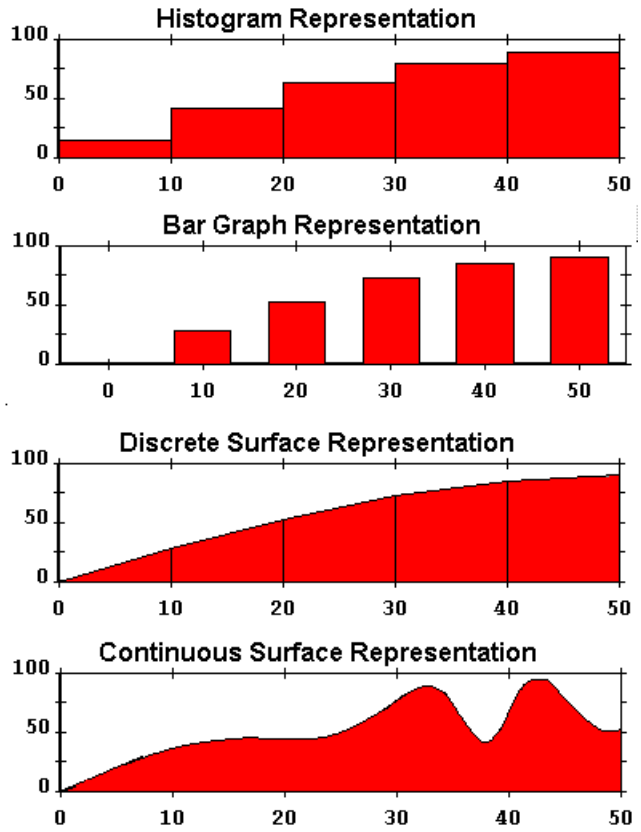

Figure 4-4: Data set representations

4.4 Histograms

The purpose of a histogram is to give the user a read-out on how often a variable or quantity has a particular value, or range of values. Each bar in the chart shows how many times the monitored quantity has taken on that value.

Histograms automatically acquire display capability, but including the special qualifier `DYNAMIC` creates histograms that are automatically updated *as the data change*, without the necessity of issuing repeated **`DISPLAY`** commands. Histogram names should not be included in a **`DISPLAY VARIABLES INCLUDE`**... statement, but are instead declared through a `TALLY` or `ACCUMULATE` statement.

```
TALLY hist.name(low TO high BY interval) AS THE [DYNAMIC]
      HISTOGRAM OF var.name
```

or

```
ACCUMULATE hist.name(low TO high BY interval) AS THE [DYNAMIC]
      HISTOGRAM OF var.name
```

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

In Figure 4.5 a histogram is shown which was obtained from the example called “bank” which simulates bank customers waiting in line for a fixed number of “teller” resources. Each bar in the histogram shows the number of “customers” that waited between (n) and (n+1) minutes for a teller, where the number of minutes (n) is shown on the x-axis.

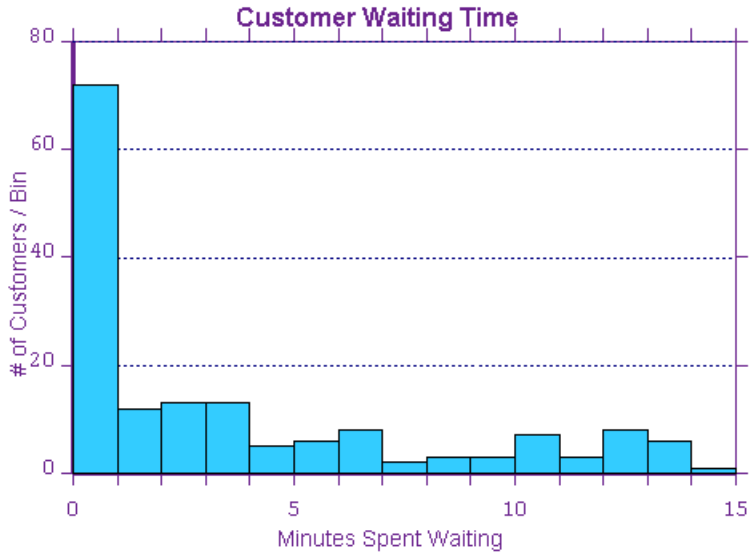

Figure 4-5:Bank model histogram.

Create a “2-D Chart” in SIMSCRIPT Studio to use as a histogram by right clicking on the “graphics.sg2” tag, selecting “new”, then picking “2-D Chart” from the list. (See Chapter 4.1). Keep in mind that each dataset in your chart will correspond to a single histogram.

In the program code, the graph is loaded and displayed using the display or show statements.

```
DISPLAY HISTOGRAM hist.name1,hist.name2,.. WITH "my_chart.grf"
```

or

```
SHOW HISTOGRAM hist.name1,hist.name2,... WITH "my_chart.grf"
```

One of the above statements must be invoked before any values are assigned to the monitored variable (and before any other reference is made to the histogram name). The first assignment to the monitored variable for dynamic histograms will cause the histogram to be displayed, and any further references will cause the chart’s appearance 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 chart. These can be edited in SIMSCRIPT Studio. Should the displayed bounds on the Y-axis be exceeded during the simulation, the histogram will rescale automatically.

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

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

Example 4.2: Show the histogram of a single variable

For this example we want to see the only the end results of the simulation run—therefore a simple (non-dynamic) histogram is used. Notice that the `show…` statement precedes the first assignment to **RANDVAR**. Assignments to display variable triggers data collection, and the graph must be loaded from "graphics.sg2" at this time. The `display…` statement at the end of the program makes the chart visible. Here are the steps to create this chart in SIMSCRIPT Studio.

1. Create the histogram chart in SIMSCRIPT Studio by right-clicking on "graphics.sg2" and selecting "new". From the dialog box choose "2D Chart" from the list, set the name to "graph2.grf", then click "Create".
2. From the graph editor window double-click on the chart. From the "Chart Properties" dialog you can set the title to something appropriate like "Random plot". Change the titles of X axis to "Value" and the Y axis title to "Count" by clicking on the "X-axis" and "Y-axis" buttons.
3. Select the top item in the list of datasets then click on the "Edit" button. From the "Dataset property" dialog, make sure that under "Representation" the "Histogram" radio button is selected, and the "Fill" box is checked. Change the "Legend" to "EXPONENTIAL.F".
4. Click on the OK button to dismiss the "Chart Properties" dialog box.

```
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 "graph2.grf" '' load from graphics file
let NSAMPLES = 50
for COUNT = 1 to NSAMPLES
    let RANDVAR = exponential.f(5.0,1)
display HISTO '' see the graph
read as /
end
```

SIMSCRIPT will allow you to show more than one histogram on the same chart graph. Assuming you have added multiple datasets to your chart in the graph editor, each of these data sets is connected to one of the histogram names included in the show/display statement.

Example 4.3: Show two dynamic histograms in the same chart

The following is a program to show two histograms in the same chart. The first histogram will show a uniform random distribution while the second shows a normal distribution.

1. Create the graph in SIMSCRIPT Studio by right-clicking on "graphics.sg2" and selecting "new". From the dialog box choose "2D Chart" from the list, set the name to "graph3.grf", then click "Create".
2. From the "Chart Properties" dialog, add another data set by clicking on the "Add" button. Also select the "Side by side" button under "Handling of Multiple Data Sets"
3. Select the top item then click on the "Edit" button. From the "Dataset Properties" dialog, make sure that under "Representation" the "Histogram" radio button is selected, and the "Fill" box is checked. Change the "Legend" to "UNIFORM.F".
4. Apply step 2 to the second dataset except change its legend to "NORMAL.F".

```
preamble
define LO, HI, DELTA, K as integer variables
define VAR.NAME1, VAR.NAME2 as a real variables
tally HIST.NAME1(LO to HI by DELTA) as the dynamic histogram of
VAR.NAME1
tally HIST.NAME2(LO to HI by DELTA) as the dynamic histogram of
VAR.NAME2
end

main
display histogram HIST.NAME1,HIST.NAME2 with "graph3.grf"
for K = 1 to 500
do
    let VAR.NAME1 = UNIFORM.F(LO, HI, 1)
    let VAR.NAME2 = NORMAL.F(LO, HI, 1)
loop
read as /
end
```

Because the histograms are specified as dynamic, they redisplay any time the variable **`VAR.NAME1`** or **`VAR.NAME2`** is assigned a value. It is not necessary to use a DISPLAY statement to update the chart.

Example 4.4: A Time-Weighted Accumulated Dynamic Histogram

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. In this example, histogram limits are declared in terms of variables. These variables obtain their values from the X-axis specification of the 2-d chart created in SIMSCRIPT Studio. To create the graph, follow the steps outlined in Example 4.3 (except call the graph "graph4.grf").

```
Preamble
define RANDVAR, LO, HI, DELTA as double variables
accumulate HIST(LO to HI by DELTA) as the dynamic histogram of RANDVAR
processes include SAMPLE
end

process SAMPLE
until TIME.V gt 100
do
    wait exponential.f(5.0, 1) units
    let RANDVAR = uniform.f(1, 10, 2)
```

```
loop
end

main
show HIST with "graph4.grf"
activate a SAMPLE now
let TIMESCALE.V = 10
start simulation
read as /
end
```

4.5 Time Trace Plots

SIMSCRIPT Studio will allow you to create a plot showing the value of a single variable plotted on the Y-axis while simulation time is plotted to the X-axis. Essentially the trace plot allows the user to see not only the current value (as in a level meter), but the value at any previous time value. Trace plots use the 2-d chart, but apply to the *display variable* and not histograms. Assuming we wanted to monitor the variable "RANDVAR" with a trace plot, the following code would be placed in the Preamble:

```
Define RANDVAR as a double variable
Display variables include RANDVAR
```

Include a trace plot by using SIMSCRIPT Studio to create a 2-D Chart. You should make the following changes to a new chart: From the "Chart Properties" dialog box, check the "trace plot" check box. Also, ensure that every dataset in the graph has the "Continuous Surface" representation.

There are two actions that can be taken by SIMSCRIPT when simulation time becomes greater than the maximum value on the chart. The first option is to "scroll" previous data to the left thus making room for more data. When **`TIME.V`** exceeds the X-axis maximum, a constant is added automatically to both the X-axis minimum and maximum. Therefore, time value data at the beginning (right side) of the plot is discarded.

If the above "scrolling window" is not appropriate, you can check the "Compress Data" box in the "X-Axis Properties" dialog box in SIMSCRIPT Studio. Under this option, SIMSCRIPT will increase only the X-axis maximum when **`TIME.V`** becomes too large. In this case, no previous data is ever discarded.

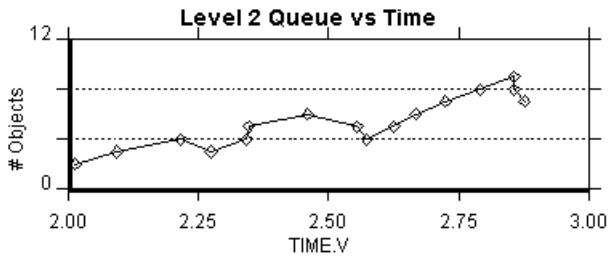

Figure 4-6:Time trace plot using the "scrolling window".

Example 4.5: Plotting a variable over time
This example is constructed as a simulation so that it can be used to illustrate the use of a time trace plot. (A dial or level meter could be substituted without changing the program code). In the program, 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 real-time second that should correspond to one unit of simulated time.

1. Create the histogram chart in SIMSCRIPT Studio by right-clicking on "graphics.sg2" and selecting "new". From the dialog box choose "2D Chart" from the list, set the name to "graph5.grf", then click "Create".
2. From the graph editor window double-click on the chart. From the "Chart Properties" dialog check the "Time Trace Plot" box.
3. Modify the X-Axis by clicking on the "X-Axis" button. Check both the "Compress data" and "Rescalable" boxes. Set the title to "TIME.V".
4. Click on the item in the "Datasets" list box and push the "Edit" button. From the "Dataset Properties" dialog box, under "Representation" select "Continuous Surface". Clear the "Fill" check box, and set the "Line" and "Interpolate" boxes.

```
Preamble
define RANDVAR as a double variable
display variables include RANDVAR
processes include SAMPLE
end

process SAMPLE
until TIME.V gt 1000
do
    let RANDVAR = uniform.f(0, 100, 2)
    wait exponential.f(5.0, 1) units
loop
end

main
show RANDVAR with "graph5.grf"
activate a SAMPLE now
let TIMESCALE.V = 10   '' 0.1 sec per unit
start simulation
read as /
end
```

4.6 Simple X-Y Plots

SIMSCRIPT can be used to generate a simple line or surface graph when there is no simulation running. This is done by simply using a time trace plot described above. To add a data point to the graph, its x coordinate is assigned to the TIME.V global. The variable used for the graph is set to the desired y value.

Example 4.6: Drawing a line graph of a simple function
In this example the function y = 100 / x+1 is plotted. For this example, borrow the same graph used in Example 4.5, (but change its name to "graph6.grf"). A temporary variable "SAVTIME" is used to preserve the original value of TIME.V in case this same code is used in a simulation.

```
Preamble
define YPLOT as a double variable
display variables include YPLOT
end

main
define SAVTIME as a double variable
let SAVTIME = TIME.V
show YPLOT with "graph6.grf"
for TIME.V = 1.0 to 100.0 do
  let YPLOT = 100.0 / TIME.V
loop
let TIME.V = SAVTIME
read as /
end
```

Example 4.7: Drawing a X-Y scatter plot

In this example, a circle of data points is drawn in a graph.

1. Create the histogram chart in SIMSCRIPT Studio by right-clicking on "graphics.sg2" and selecting "new". From the dialog box choose "2D Chart" from the list, set the name to "graph7.grf", then click "Create".
2. From the graph editor window double-click on the chart. From the "Chart Properties" dialog check the "Time Trace Plot" box.
3. Click on the item in the "Datasets" list box and push the "Edit" button. From the "Dataset Properties" dialog box, under "Representation" select "Continuous Surface". Clear the "Fill", and "Line" check boxes, and set the "Mark" and "Interpolate" boxes. Set the "Mark Style" to "Square".

```
Preamble
define YPLOT as a double variable
display variables include YPLOT
end

main
define THETA as a integer variable
show YPLOT with "graph7.grf"
for THETA = 0 to 360 by 10 do
  let TIME.V = 40 * COS.F(2.0 * THETA * PI.C / 360.0) + 50
  let YPLOT = 40 * SIN.F(2.0 * THETA * PI.C / 360.0) + 50
loop
read as /
end
```

4.7 Clocks

SIMSCRIPT provides both analog (circular with hands) and digital clocks that can be used to show current simulation time. The clocks are dynamic in nature and update automatically whenever the display variable (containing time) changes in value.

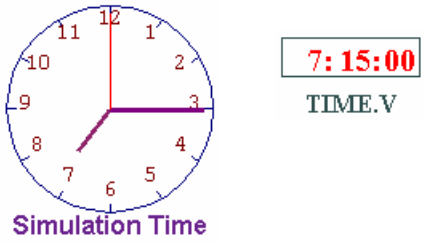

Figure 4-7: Clocks

4.7.1 Editing a Clock in SIMSCRIPT Studio

Create a clock by selecting either "Analog Clock" or "Digital Clock" from the list in the "Create New Graphic" dialog box. When you click on the "Create" button a graph editor window will appear containing the clock. (See Chapter 4.1). 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 on the right. Double-click on the clock image to show the "Clock Properties" dialog box containing the following:

- **Name** – The name of the object within the current graphics library. Use this name in your program via the "Show" or "Display" statement.
- **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 a borders around the face of the clock.

4.7.2 Adding a Clock to Your Program

A clock in SIMSCRIPT works by showing the current value of a display variable (See Chapter 4.2.2). You define this variable in your PREAMBLE like this:

```
Define CLOCKTIME as a double variable
Display variables include CLOCKTIME
```

But there needs to be some way to automatically update the clock whenever simulation time changes. This can be done by writing a time synchronization routine and assigning it to the **TIMESYNC.V** system variable. This routine will then be called whenever SIMSCRIPT is about to update the value of **TIME.V** and will allow you to change the clock by assigning your display variable to the given parameter called "TIME".

```
Let TIMESYNC.V = 'CLOCK.UPDATE'
...
Routine CLOCK.UPDATE given TIME yielding NEWTIME
Define TIME, NEWTIME as double variables
Let NEWTIME = TIME
Let CLOCKTIME = TIME '' update the clock
End
```

By updating the display variable in this routine, you ensure that the clock will be updating regardless of how many processes are happening in the simulation.

Example 4.8: Showing Simulation Time in a Clock
For this example, create either an analog or digital clock in SIMSCRIPT Studio as described above. Set its name to "graph8.grf". This program will run for 100 seconds and update a clock as it goes.

```
Preamble
Define CLOCKTIME as a double variable
Display variables include CLOCKTIME
Processes include SAMPLE
End

Process SAMPLE
wait 20.0 / (24. * 60.) units '' wait 20 simulated minutes
end

Routine CLOCK.UPDATE given TIME yielding NEWTIME
Define TIME, NEWTIME as double variables
Let NEWTIME = TIME
Let CLOCKTIME = TIME '' update the clock (time in days)
End

Main
show CLOCKTIME with "graph8.grf"
activate a SAMPLE now
let TIMESCALE.V = 100 * 24 * 60 '' 1 real sec. per sim. minute
Let TIMESYNC.V = 'CLOCK.UPDATE'
start simulation
read as /
End
```

4.8 Pie Charts

A pie chart is useful for displaying the fraction of the whole that each portion of a population takes up. Each sector has an area in proportion to its fraction. For example, if a quantity is one-third of the whole, its slice will be one-third of the pie chart.

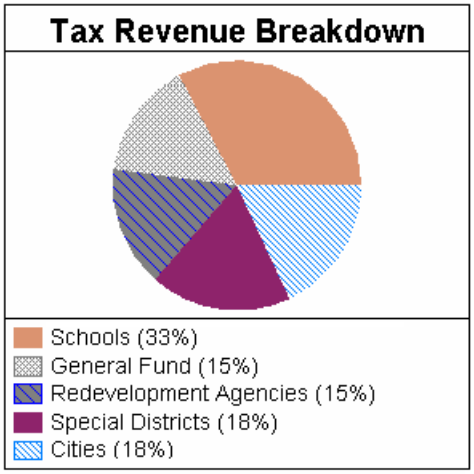

Figure 4-8: A typical pie-chart shown in SIMSCRIPT.

4.8.1 Editing a Pie chart in SIMSCRIPT Studio

Add a pie chart to your application by right clicking on the "graphics.sg2" tag then selecting "new" from the popup menu. Select "Pie chart" from the list-box then click on the "Create" button. Position your pie chart by dragging it with the mouse. It can be resized by dragging one of the small green rectangles at a corner. The color or fill style of a slice in the chart can be changed by selecting it, then choosing from the style palette on the right. You can also select the text labels and borders and change the style, color and font by using the palette on the right edge of the window.

Double click on the piechart to display a dialog box showing its properties. From this dialog you can add and remove slices in the chart, or change the name displayed in the legend. This dialog box contains the following items:

- **Name** – The name of the object within the current graphics library. This name is used in you program code.
- **Title** – Text of title displayed on top.
- **Show border** – 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 click on the **Add** button.
2. To remove a slice, select its name in the list box and click on 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.

4.8.2 Adding a Pie Chart to Your Program

Basically, a pie chart can be used to monitor a 1 dimensional array in SIMSCRIPT. Each element in the array contains the value for one of the slices in the pie. (Naturally you should make sure that the number of elements in the array matches the number of slices added in SIMSCRIPT Studio.)

```
Define PIE.VALUES as a 1-dim double array
Display variables include PIE.VALUES
```

The array should be defined as a *display variable* in the Preamble. After the pie chart has been loaded using a "SHOW" statement, assignments made to any element in the array will update the pie chart automatically.

Example 4.9: A Dynamic Pie chart
Here we show a pie chart with four slices. As the program runs, an array with 4 elements is assigned values repeatedly, causing the chart to be redisplayed. To construct the chart in SIMSCRIPT Studio:

1) Add a new pie chart named "graph9.grf" to the project.
2) Double click on the pie chart to show its properties.
3) Click on the first slice shown in the list box. Enter "y = sqrt(TIME.V)" in the "Slice name" field then click on the "Update" button.
4) Click on the second slice shown in the list box. Enter "y = TIME.V" in the "Slice name" field then click on the "Update" button.
5) Click on the third slice shown in the list box. Enter "y = TIME.V * TIME.V" in the "Slice name" field then click on the "Update" button.
6) Click on the "Add" button to create a new slice. Select the new fourth item in the list. Enter "y=TIME.V * TIME.V * TIME.V" in the "Slice name" field then click on the "Update" button.

```
Preamble
Define Y as a 1-dim double array
Display variables include Y
Processes include PIECHART
End

Process PIECHART
While 1 eq 1 do
    '' assign values to the slices, then wait
    Let Y(1) = SQRT.F(TIME.V)
    Let Y(2) = TIME.V
    Let Y(3) = TIME.V * TIME.V
    Let Y(4) = TIME.V * TIME.V * TIME.V
    Display Y
    Wait 0.01 units
Loop
End

Main
Reserve Y(*) as 4
Show Y with "graph9.grf"
Let TIMESCALE.V = 100  '' 1 sec per unit
```

```
Activate a PIECHART now
Start simulation
End
```

Example 4.10: 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 histogram. The code and graphics for the complete model are included on the distribution kit for SIMSCRIPT II.5. The simulation model is described in the `Preamble, Main`, the `INITIALIZE` routines, and in the `GENERATOR` and `CUSTOMER` processes. The 2-d histogram is defined in lines 12 to 13 of the Preamble, and initialized in line 9 of **`INITIALIZE.GRAPHICS`**. The animated clock is defined in lines 9-10 of the Preamble, created in line 5 of **`INITIALIZE.GRAPHICS`**, and updated in line 4 of **`CLOCK.UPDATE`**. In addition to the model enhancements, SIMSCRIPT Studio was used to produce the three graphs to be used for display purposes. These are: **`clock.grf, queue.grf`**, and **`wait.grf`**.

```
1     preamble '' BANK - Modernizing a Bank
2     normally, mode is undefined
3     processes include GENERATOR and CUSTOMER
4     resources include TELLER
5     define NO.OF.TELLERS as an integer variable
6     define MEAN.INTERARRIVAL.TIME, MEAN.SERVICE.TIME, DAY.LENGTH
7       and WAITING.TIME as real variables
8     Define WLO, WHI and WDELTA as integer variables
9     Define CLOCKTIME as a double variable
10    Display variables include CLOCKTIME, N.Q.TELLER
11    Graphic entities include SHAPE
12    Tally WAITING.TIME.HISTOGRAM (WLO to WHI by WDELTA)
13      as the dynamic histogram of WAITING.TIME
14    end ''preamble

1     main
2     call INITIALIZE
3     call INITIALIZE.GRAPHICS
4     start simulation
5     read as /
6     end ''main

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

1     routine INITIALIZE.GRAPHICS
2     Define DEVICE.ID and TITLE as pointer variables
3     Let timescale.v = 1000 '' clock ticks (1/100 sec) per unit
4     Let timesync.v = 'CLOCK.UPDATE'
5     Display CLOCKTIME with "clock.grf"
6     create a SHAPE called TITLE
7     display TITLE with "title" at (15000.0, 21000.0)
8     Display N.Q.TELLER(1) with "queue.grf"
```

```
9     Display histogram WAITING.TIME.HISTOGRAM with "wait.grf"
10    end ''INITIALIZE.GRAPHICS

1     routine CLOCK.UPDATE given TIME yielding NEWTIME
2     Define TIME, NEWTIME as double variables
3     Let NEWTIME = TIME
4     Let CLOCKTIME = TIME
5     End ''CLOCK.UPDATE

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

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

5. Dialog Boxes

Dialog boxes provide an interactive way for the user to enter input data. A dialog box is a window containing a variety of input controls including buttons, text labels, tabular controls, single and multi-line text, combo, value, list, radio, and check boxes. These items will be referred to as the "fields". Tabbed dialogs can also be created in SIMSCRIPT.

Dialog boxes are constructed using SIMSCRIPT Studio's *dialog editor*. The Studio allows you to create a template for each dialog box needed by your program. Templates are saved in the file "graphics.sg2".

Routines in the SIMSCRIPT runtime library will allow you to load a dialog box, initialize its fields, display it, and also retrieve data at any time. You can also provide a "control routine" that gets called automatically whenever the user presses buttons or changes field values.

5.1 Using the Dialog Box Editor

The first step to adding a dialog box to the SIMSCRIPT program is to define its layout using the SIMSCRIPT Studio dialog box editor. To create the dialog and add it to your project, right click on the graphics library ("graphics.sg2" in the project tree) then select the "New" from the popup menu. Select "Dialog box" from the list and click OK. To edit an existing dialog box, double-click its name shown in the contents pane under "graphics.sg2".

The dialog box editor window is designed to resemble the dialog box being edited. The dialog box editor can be resized by dragging its right or bottom edges with the mouse. Double-clicking in the background of the window allows properties such as the title and library name to be specified. The library name will be used by your program when it needs to create this dialog box. As a convention, dialog box names should end in ".frm".

The new dialog box will initially contain two buttons labeled "OK" and "Cancel". Add input fields to the dialog by clicking on one of the palette buttons attached to the right side of the Studio's main window. A field can then be positioned by dragging it with the mouse. The field that is currently selected is shown with a checkered border. Click and drag any edge of the border to resize the field. You can make multiple selections by holding down the shift key when you click on a field or dragging a box over a region containing fields to be selected.

To set the label or other attribute of the field, double-click on it to display its "Properties". Each field should be given a unique "Field Name". This is important, because the field name allows your program to access the field's data. (See Table 4.1 for

detailed descriptions of the various input fields and how to edit them in SIMSCRIPT Studio.)

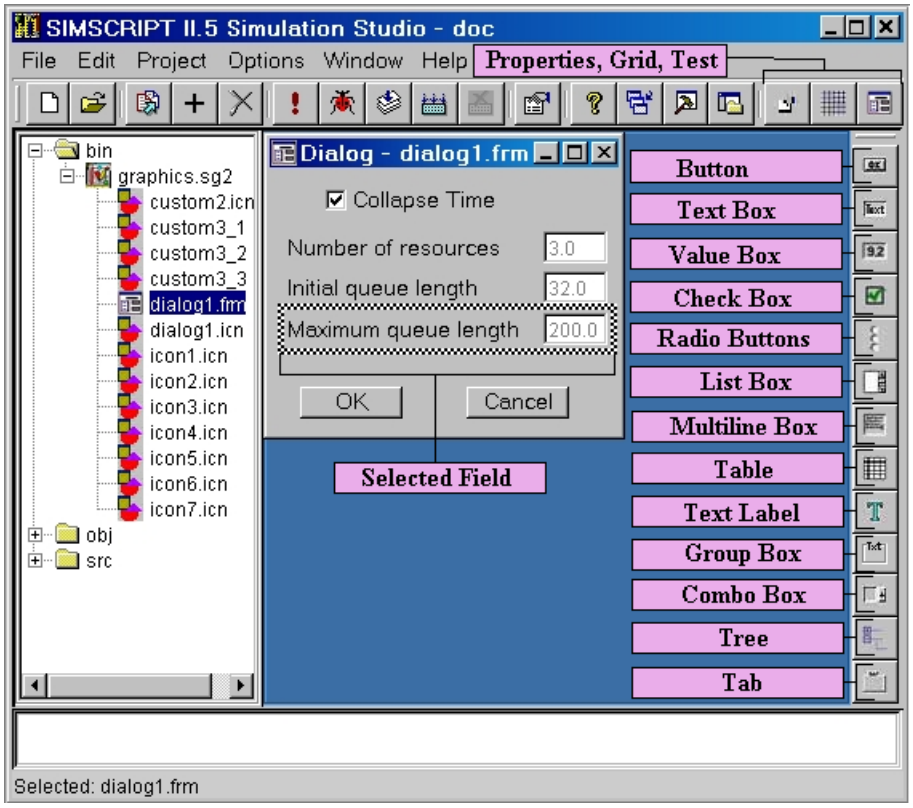

Figure 5-1:The dialog box editor

5.2 Showing a Dialog Box in SIMSCRIPT

SIMSCRIPT provides some easy to use constructs which will allow you to load and display your dialog box. The SIMSCRIPT "show" will create a new instance of a dialog box and assign it to a pointer variable, but not yet make the dialog box visible.

```
Show DIALOG.PTR with "my_dialog_box.frm"
```

Display this dialog box using the **ACCEPT.F** function as follows:

```
Let FIELD.NAME = ACCEPT.F(DIALOG.PTR, 0)
```

This function will actually display the dialog box on the screen. **ACCEPT.F** does not return until the dialog box is dismissed. At this point, it will return with the field name (text value) of the button that was clicked on to close the dialog box.

Here is a small example program that simply loads and displays a dialog box created in SIMSCRIPT Studio.

Example 5.1:

```
Main
'' - Display a dialog box.  To create this in SIMSCRIPT STUDIO:
'' 1) Right click on "graphics.sg2", select new from the popup menu
'' 2) Chose "Dialog Box" from list then click on "Create"
'' 3) Double click on the window to show the Dialog Box properties.
'' 4) Set library name to "my_dialog_box.frm", click "OK"
'' 5) Build your project

Define DIALOG.PTR as a pointer variable
Define FIELD.ID as a text variable

Show DIALOG.PTR with "my_dialog_box.frm"  '' load the dialog box
Let FIELD.ID = ACCEPT.F(DIALOG.PTR, 0)    '' display the dialog box

End
```

5.3 Setting and Accessing Field Values

Each item in you dialog box should be given a unique field name. (To set the field name, double-click on an item from the SIMSCRIPT STUDIO dialog editor window to display its properties, then fill in the "Field Name" box.) Your program can retrieve a pointer to any one of the fields in your dialog box by passing its field name the `DFIELD.F` function.

The entity attributes **`DDVAL.A`**,**`DARY.A`** and **`DTVAL.A`** of the resulting pointer will contain the numerical, array, or text data shown in the dialog. Before calling ACCEPT.F, you should initialize the fields in your dialog box by setting attributes of the field pointer. For example:

```
Let DTVAL.A(DFIELD.F("my_text_box_field", DIALOG.PTR)) = TEXT.VALUE
Let DDVAL.A(DFIELD.F("my_value_box_field", DIALOG.PTR)) = REAL.VALUE
Let DARY.A(DFIELD.F("my_list_box_field", DIALOG.PTR)) =TEXT.ARRAY(*)
```

Use these same attributes to get the data after dialog interaction is complete and ACCEPT.F has returned.

```
Let TEXT.VALUE = DTVAL.A(DFIELD.F("my_text_box_field", DIALOG.PTR))
Let REAL.VALUE = DDVAL.A(DFIELD.F("my_value_box_field", DIALOG.PTR))
Let TEXT.ARRAY(*) = DARY.A(DFIELD.F("my_list_box_field", DIALOG.PTR))
```

Example 5.2:

```
Main
'' - Display a dialog with a value box.  To create in SIMSCRIPT STUDIO:
'' 1) Double-click on "my_dialog_box.frm" (create in Example 5.1)
'' 2) Click on the value box tool button on the right palette
'' 3) Position the value box by dragging it with mouse
'' 4) Double-click on the value box to display properties.
'' 5) Change Field name to "my_value_box_field"
```

```
'' 6) Click OK, Build the project

Define DIALOG.PTR as a pointer variable
Define FIELD.NAME as a text variable
Define REAL.VALUE as a real variable
Let REAL.VALUE = 12.3

'' Create a new dialog box and load it from template library
Show DIALOG.PTR with "my_dialog_box.frm"

'' initialize the dialog box
Let DDVAL.A(DFIELD.F("my_value_box_field", DIALOG.PTR)) = REAL.VALUE

'' Display the dialog box and wait for terminating button click
Let FIELD.NAME = ACCEPT.F(DIALOG.PTR, 0)

'' Get the data from dialog
Let REAL.VALUE = DDVAL.A(DFIELD.F("my_value_box_field", DIALOG.PTR))
List REAL.VALUE

End
```

Consult the following table to determine which of the attribute to use on a given field:

Table 5-1:How to access various fields using display entity attributes.

FIELD	Data Application	User Can Edit	Calls Control Routine	Entity Attributes
Button	No data	No	Yes	Not used
Check box	Receive yes / no input	Yes	Yes	DDVAL.A
Combo box	Select from a list	Yes	Yes	DTVAL.A, DDVAL.A, DARY.A
Label	Single text or numeric value	No	No	DTVAL.A, DDVAL.A
List box	Select from a list	No	Yes	DTVAL.A, DDVAL.A, DARY.A
Multi-line text box	Display many lines of text	Yes	No	DARY.A
Progress bar	Single numeric value	No	No	DDVAL.A
Radio button	Select one of many	Yes	Yes	DDVAL.A
Table	Select one from a 2-d grid	No	Yes	DTVAL.A, DARY.A, DDVAL.A
Text box	Enter a single line of text	Yes	Maybe	DTVAL.A
Tree	Select from hierarchical tree	No	Yes	DTVAL.A, DDVAL.A, DARY.A
Value box	Enter a single numeral	Yes	Maybe	DDVAL.A

5.4 Using Control Routines to get Input Notification

In some cases, you will want to provide code to immediately handle dialog input events generated by the user (such as button clicks, text box modification, etc) while the dialog is still visible. A control routine can be passed to **`ACCEPT.F`** that will be called whenever a user performs some action on one of the items in a dialog box. This is a useful way to perform immediate validation or cross-checking of fields. Your program should define the routine like this:

```
routine DIALOG.RTN given FIELD.NAME, DIALOG.PTR yielding
FIELD.STATUS

define FIELD.NAME as text variable       '' name of field
define DIALOG.PTR as pointer variable    '' pointer to dialog
define FIELD.STATUS as integer variable  '' set this to -1,0,1
```

The routine name is passed to ACCEPT.F like this:

```
let FIELD.NAME = ACCEPT.F(DIALOG.PTR, 'DIALOG.RTN')
```

Consult Table 5.1 or Section 5.3 to see which items generate calls to the control routine. To get a callback from a text or value box, mark "Return Selectable" in the "properties" dialog box for the item.

Handler code for the control routine should check the FIELD.NAME parameter to determine which of the items in the dialog was clicked on or changed. Before the dialog is first displayed, the control routine is called with FIELD.NAME equal to "INITIALIZE". In addition, your control routine will be called with FIELD.NAME equal to "BACKGROUND" if the user clicks in the canvas of the graphics window (not dialog box window). You can retrieve the location of this mouse click through the **LOCATION.A** attribute of the display entity pointer **DINPUT.V** (if nonzero).

The **FIELD.STATUS** parameter can be used to communicate the following to **ACCEPT.F:**
0 -- Accept the input
1 -- Terminate the interaction (return from ACCEPT.F).

Here is a small program that uses a control routine:

```
Routine DIALOG.RTN given FIELD.NAME, DIALOG.PTR yielding FIELD.STATUS
'' - Use control routine to receive callbacks from SIMSCRIPT
'' To create in SIMSCRIPT STUDIO:
'' 1) Double-click on "my_dialog_box.frm" (create in Example 5.1,2)
'' 2) Double-click on the value box to display its Properties
'' 3) Position the value box by dragging it with mouse
'' 4) Double-click on the value box to display properties.
'' 5) Change Field name to "my_value_box_field"
'' 6) Click OK, Build the project

define FIELD.NAME as text variable       '' name of field
define DIALOG.PTR as pointer variable    '' pointer to dialog
define FIELD.STATUS as an integer variable '' set to -1,0,1

Select case FIELD.NAME
Case "INITIALIZE"
      let DDVAL.A(DFIELD.F("my_value_box_field", DIALOG.PTR)) = 123.4
Case "value_box_field_name"
      List DDVAL.A(DFIELD.F("my_value_box_field", DIALOG.PTR))
default
Endselect
```

```
Let FIELD.STATUS = 0   '' everything is OK!

Return
End

Main
'' assumes we have created a dialog box named "my_dialog_box.frm"
'' containing a value box named "my_value_box_field"

Define DIALOG.PTR as a pointer variable
Define FIELD.NAME as a text variable

'' Load and then show the dialog
Show DIALOG.PTR with "my_dialog_box.frm"
Let FIELD.NAME = ACCEPT.F(DIALOG.PTR, 'DIALOG.RTN')

'' Show the result
List DDVAL.A(DFIELD.F("my_value_box_field", DIALOG.PTR))

End
```

5.5 Enable and Disable fields

In many cases, you may want to activate and deactivate items in your dialog box in response to the user changing one of the fields. This can be done inside your control routine. The routine **`SET.ACTIVATION.R`** can be used to disable or enable a field. The syntax is:

```
Call SET.ACTIVATION.R given FIELD.PTR, ACTIVATION.STATUS
''  ACTIVATION.STATUS = 1 means to enable, 0 means disable
```

For example, suppose you want to deactivate the field "value_box_field" when user clicks on "my_check_box_field". The control routine should now contain the code:

```
Select case FIELD.NAME
When "my_check_box_field"
      Call SET.ACTIVATION.R(
      DFIELD.F("my_value_box_field", DIALOG.PTR),
      DDVAL.A(DFIELD.F("my_check_box_field", DIALOG.PTR)))
      let DDVAL.A(DFIELD.F("my_value_box_field", DIALOG.PTR)) =
123.4
. . .
```

It is also possible to hide and show fields using the SIMSCRIPT "display" and "erase" statements. i.e.

```
Erase DFIELD.F("my_value_box_field", DIALOG.PTR)
Display DFIELD.F("my_value_box_field", DIALOG.PTR)
```

5.6 Dialog Boxes: Field Types

SIMSCRIPT allows you to use a variety of fields in you dialog boxes. Fields are added using SIMSCRIPT Studio. Refer to Figure 5.1 to see which button to click on to add a particular field. Figure 5.2 below shows an example of a dialog box containing all types of fields:

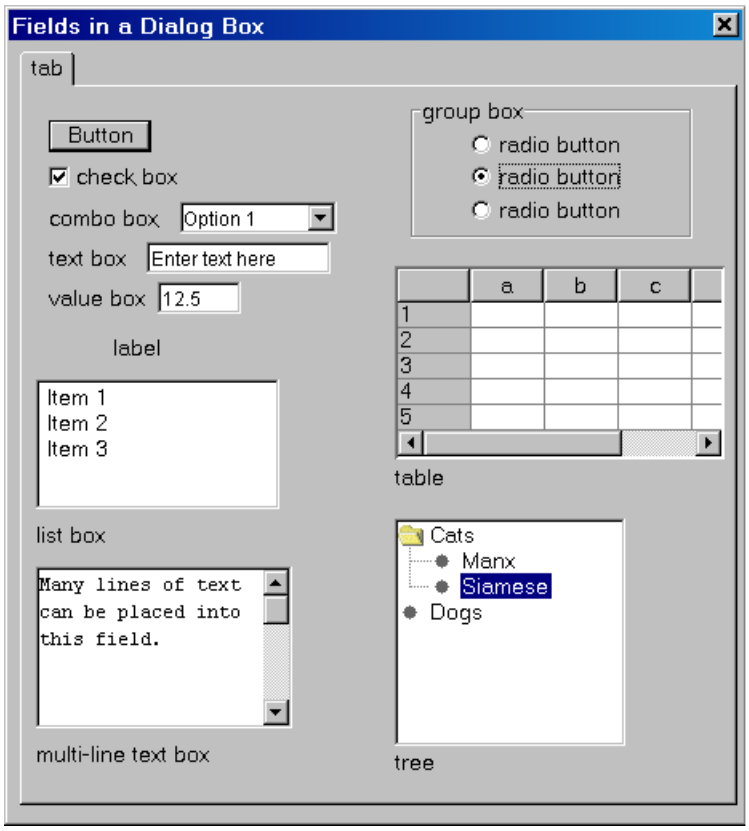

Table 5-2: Dialog box containing many fields.

5.6.1 Buttons

Buttons do not hold data, but it can be used to activate a control routine, verify contents of value boxes, or close its dialog box. In SIMSCRIPT Studio, double click on a button to edit its properties.

- **Label** – Text shown on the face of the button.
- **Default** – Setting this item will make this button the "default" button. Default buttons can be "pressed" using the <Enter> key.
- **Verifying** – When a "Verifying" button is clicked on, each numerical value in the dialog will be checked against its acceptable range. (see "Value Box").
- **Terminating** – Clicking on a "Terminating" button will erase its dialog. If the dialog box is modal, the application will return immediately from `ACCEPT.F`.

5.6.2 Check Box

A check box is used to receive and show yes/no input. The SIMSCRIPT Studio "Checkbox Properties" dialog will show the following:

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

In your program, the check box is checked when `DDVAL.A` = 1, and cleared when `DDVAL.A` = 0.

5.6.3 Combo Box

A combo box is of a text box with a drop down list attached to it right side. SIMSCRIPT Studio lets you set the following properties:

- **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.
- **Sorted Alphabetically** – Indicates whether or not to sort the items automatically.
- **Selectable using return** – The control routine is also invoked when the user presses the **Return** key.

`DARY.A` contains a selectable list of alternatives for the text box. Your program can also use **`DTVAL.A`** to set the text in the box. The control routine is called whenever the user selects an item from the list. The routine can use the **`DTVAL.A`** attribute to determine which item was selected.

Example 5.4: Use a combo box in a dialog

```
Main
''  Display a dialog with a value box.  To create in SIMSCRIPT STUDIO:
'' 1) Create a new dialog called "my_dialog_box.frm" (see Example 5.1)
'' 2) Click on the combo box tool button on the right palette
'' 3) Position the combo box by dragging it with mouse
'' 4) Double-click on the combo box to display properties.
'' 5) Change Field name to "my_combo_box_field"
'' 6) Click OK, Build the project

Define DIALOG.PTR as a pointer variable
Define FIELD.NAME as a text variable

Define COMBO.ARRAY as a 1-dim text array
Reserve COMBO.ARRAY(*) as 3
Let COMBO.ARRAY(1) = "Hello"
Let COMBO.ARRAY(2) = "There"
Let COMBO.ARRAY(3) = "World"

'' Create a new dialog box and load it from template library
```

```
Show DIALOG.PTR with "my_dialog_box.frm"

'' initialize the dialog box
Let DARY.A(DFIELD.F("my_combo_box_field", DIALOG.PTR)) = COMBO.ARRAY(*)

'' set the initial text in the box
Let DTVAL.A(DFIELD.F("my_combo_box_field", DIALOG.PTR)) = "World"

'' Display the dialog box and wait for terminating button click
Let FIELD.NAME = ACCEPT.F(DIALOG.PTR, 0)

End
```

5.6.4 Labels & Group Boxes

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. Labels can have an attached *group box* which shows the user a related set of fields. Through SIMSCRIPT Studio you can specify the following properties:

- **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_field", FORM.PTR)) = "Hello World"
```

or

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

The control routine is not called when the user clicks on a label.

Example 5.5: Setting static (read only) text and numerical labels

```
Main
''  Display a dialog some static text.  To create in SIMSCRIPT STUDIO:
'' 1) Create a dialog box entitled "my_dialog_box.frm" (see Example 5.1)
```

```
'' 2) Click on the label tool button on the right palette three times
'' 3) Position each label control by dragging it with mouse
'' 4) Double-click on each label to display properties.
'' 5) Call a label  "my_label_field1". Select "Normal Text" radio button
'' 6) Call a label  "my_label_field2". Select "Truncated Text" radio button.
''     Set the field width to 9
'' 7) Call a label  "my_label_field1". Select "Formatted Real" radio button
''     Set the field width to 5, precision to 2
'' 8) Click OK, Build the project

Define DIALOG.PTR as a pointer variable
Define FIELD.NAME as a text variable

'' Create a new dialog box and load it from template library
Show DIALOG.PTR with "my_dialog_box.frm"

'' initialize the dialog box
Let DTVAL.A(DFIELD.F("my_label_field1", DIALOG.PTR)) = "Hello World"
Let DDVAL.A(DFIELD.F("my_label_field2", DIALOG.PTR)) = "Truncated Text"
Let DDVAL.A(DFIELD.F("my_label_field3, "DIALOG.PTR)) = 33.333333

'' Display the dialog box and wait for terminating button click
Let FIELD.NAME = ACCEPT.F(DIALOG.PTR, 0)

End
```

5.6.5 List Box

A list box is used to accept input from a sequential list of text items. The list may vary in length and may contain scrollbars. You can define the list to accept only single item selections, or accept multiple item selections using the **Shift** and/or **Ctrl** keys. A list box has the following properties:

- **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.

In your program, use the **`DARY.A`** attribute to set the array containing the list of text values, and the **`DDVAL.A`** attribute to set the index of the initially selected item.

The control routine is called when a selection is made. The program can then examine the DDVAL.A attribute to get the index of the newly selected item. DTVAL.A will contain the text of the selected item.

List boxes can be defined in SIMSCRIPT Studio to allow multiple selections. The user can add selections to the list box by holding down the <shift> key while clicking. The control routine will be called each time any item changes selection state. The routine **`LISTBOX.SELECTED.R`** can be used to get all the selected items or even to determine if the user has **double-clicked** on an item. 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 if the item is not selected.

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

```
Let DDVAL.A(DFIELD.F("my_listbox_field", DIALOG.PTR)) = 0
Display DFIELD.F("my_listbox_field", DIALOG.PTR)
```

Example 5.6: Multiple selection list box.

```
Main
'' Show a dialog containing a list box.  To create in SIMSCRIPT STUDIO:
'' 1) Create a new dialog named "my_dialog_box.frm" (see Example 5.1)
'' 2) Click on the list box tool button on the right palette
'' 3) Position the list box by dragging it with mouse
'' 4) Double-click on the value box to display properties.
'' 5) Change Field name to "my_listbox_field"
'' 6) Mark the "Allow multiple selections" box.
'' 7) Click OK, Build the project

Define DIALOG.PTR as a pointer variable
Define LIST.ARRAY as a 1-dim text array
Define ITEM.SELECTED.FLAG, I as integer variables
Define FIELD.ID as a text variable

'' Create a new dialog box and load it from template library
Show DIALOG.PTR with "my_dialog_box.frm"

'' initialize the dialog box
Reserve LIST.ARRAY(*) as 3
Let LIST.ARRAY(1) = "Hello"
Let LIST.ARRAY(2) = "There"
Let LIST.ARRAY(3) = "World"
Let DARY.A(DFIELD.F("my_listbox_field", DIALOG.PTR)) = LIST.ARRAY(*)

'' Display the dialog box and wait for terminating button clicklet
Let FIELD.ID = ACCEPT.F(DIALOG.PTR, 0)

'' Get the selection state for each item in the list
for I = 1 to DIM.F(LIST.ARRAY(*))
do
    call LISTBOX.SELECTED.R
        given DFIELD.F("my_listbox_field", DIALOG.PTR), I
        yielding ITEM.SELECTED.FLAG

    select case ITEM.SELECTED.FLAG
    case 0
      write LIST.ARRAY(I) as "ITEM ", T *, " not selected. ", /
    case 1
      write LIST.ARRAY(I) as "ITEM ", T *, " was selected !", /
    case 2
      write LIST.ARRAY(I) as "ITEM ", T *, " was double-clicked!", /
    endselect
loop

'' wait for user to hit the <return> key
Read as /

End
```

5.6.6 Multi-line Text Box

A multi-line text box is useful for displaying or allowing a user to edit multiple lines of text. Horizontal and vertical scroll bars are attached if needed. SIMSCRIPT Studio allows you to change the following properties:

- **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 any line of text is too long to be viewed in the text box. If not checked, long lines of text may appear truncated if unable to fit inside the edit box.

Use the **DARY.A** attribute to set/get lines of text. Note that the control routine is NOT called when the **Return** key is pressed. Here is some example code that sets the initial text, receives input, then displays lines of text entered by the user:

Example 5.7: Multi-line Text Box

```
Main
''  Display a dialog containing an edit box.  To create in SIMSCRIPT
STUDIO:
'' 1) Create a dialog box called "my_dialog_box.frm" (see Example 5.1)
'' 2) Click on the multi-line text box tool button on the right palette
'' 3) Position the edit box by dragging it with mouse
'' 4) Double-click on the edit box to display properties.
'' 5) Change Field name to "my_edit_box_field"
'' 6) Click OK, Build the project

Define DIALOG.PTR as a pointer variable
Define INITIAL.TEXT, FINAL.TEXT as a 1-dim text array
Define I as an integer variable

'' load the dialog box
Show DIALOG.PTR with "my_dialog_box.frm"

'' Initialize the multi-line box with field name "my_edit_box_field"
Reserve INITIAL.TEXT (*) as 2
Let INITIAL.TEXT (1) = "First line of text"
Let INITIAL.TEXT (2) = "Second line of text"
Let DARY.A(DFIELD.F("my_edit_box_field", DIALOG.PTR)) = INITIAL.TEXT(*)

'' Show the dialog box, wait for input
If ACCEPT.F(DIALOG.PTR, 0) eq "OK"
  '' Get and display the text that the user has entered
  Let FINAL.TEXT(*) = DARY.A(DFIELD.F("my_edit_box_field", DIALOG.PTR))
  For I = 1 to DIM.F(FINAL.TEXT(*)) do
    List FINAL.TEXT(I)
  Loop
Always

'' Wait for user to hit return
```

```
Read as /

End
```

5.6.7 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. You can set the minimum and maximum values in SIMSCRIPT Studio. The user cannot change the position of the bar with the mouse. The following properties can be specified in SIMSCRIPT Studio.

- **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.

Usually progress bars need to be displayed in modeless dialog boxes. The **`ACCEPT.F`** function must return immediately without dismissing the dialog to allow your program to perform necessary processing. To make dialog box modeless, edit its properties in SIMSCRIPT Studio. Un-check "modal interaction", then set the "Don't wait" radio button in the "Action taken by ACCEPT.F" radio box.

5.6.8 Radio Box

The radio box accepts input from a fixed list of mutually exclusive toggle buttons. From SIMSCRIPT Studio, the properties dialog allows you to add and remove radio buttons from the radio box.

- To add a button, enter its *label*, and *field 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, and field name, and then press the **Update** button.

Each individual radio button in the box has its own field name, and can therefore be accessed by your program. Provide the field name to **`DFIELD.F`** then use the **`DDVAL.A`** attribute to turn the button on (**`DDVAL.A`** = 1) or off (**`DDVAL.A`** = 0). The control routine is called whenever the user clicks on a radio button. Here is an example of how to set and get the values of radio button fields.

Example 5.8: Using a radio box

```
Main
''  Display dialog with radio buttons.  To create in SIMSCRIPT STUDIO:
'' 1) Create a dialog named "my_dialog_box.frm" (see Example 5.1)
'' 2) Click on the radio box tool button on the right palette
'' 3) Position the radio button by dragging it with mouse
'' 4) Double-click on the button to display properties.
'' 5) Change Field name to "my_radio_box_field"
'' 6) Add buttons with field names "green_button", "red_button"
''    and "blue_button".
'' 7) Click OK, Build the project

Define DIALOG.PTR as a pointer variable

Show DIALOG.PTR with "my_dialog_box.frm"

'' Make the blue button initially selected
Let DDVAL.A(DFIELD.F("red_button", DIALOG.PTR)) = 0
Let DDVAL.A(DFIELD.F("green_button", DIALOG.PTR)) = 0
Let DDVAL.A(DFIELD.F("blue_button", DIALOG.PTR)) = 1

'' Display the dialog box
If ACCEPT.F(DIALOG.PTR, 0) eq "OK"
  If DDVAL.A(DFIELD.F("red_button", DIALOG.PTR)) ne 0
    Write as "Red was selected", /
  Always
  If DDVAL.A(DFIELD.F("green_button", DIALOG.PTR)) ne 0
    Write as "Green was selected", /
  Always
  If DDVAL.A(DFIELD.F("blue_button", DIALOG.PTR)) ne 0
    Write as "Blue was selected", /
  Always
Always

'' Wait for the user
Read as /

End
```

5.6.9 Table

A table is a two dimensional arrangement of selectable text fields or “cells”. The table can be scrolled both horizontally and vertically. 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. The headers are fixed and will remain in view when the table is scrolled.

You can navigate through a table using the left-, right-, up- and down-arrow keys. The callback routine is invoked whenever a cell is clicked on or an arrow key is used to move to activate a new cell. The table can automatically add a new row of cells at the bottom when the user attempts to move below the last row. Use SIMSCRIPT Studio to specify the following properties:

- **Viewed Width** – The total width in font units of the table including row headers and scroll bar.
- **Viewed Height** – The total height in font units of the 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 to the left of the cells.
- **Automatic Grow** – The table will automatically add a row if the user attempts 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**.

Your SIMSCRIPT program can initialize the table using the **DARY.A** attribute. Text values are laid out in row major order. For example, assume the table has '.NUM.COLUMNS' columns (including row headers) and '.NUM.ROWS' rows. The index into **DARY.A** for cell (CELL.COLUMN, CELL.ROW) is computed as follows:

```
Let TABLE.VALUES((CELL.ROW-1) * .NUM.COLUMNS + CELL.COLUMN) = 3
```

Note that the array containing these text values must be reserved big enough to hold all the cells. The **DDVAL.A** attribute contains the index of the currently selected cell while **DTVAL.A** contains the text for this cell. You can compute the selected row and column from **DDVAL.A** as follows:

```
Let TABLE.PTR = DFIELD.F(("table_field", DIALOG.PTR)
Let SELECTED.ROW = DIV.F(DDVAL.A(TABLE.PTR))-1, NUM.COLUMNS) + 1
Let SELECTED.COL = MOD.F(DDVAL.A(TABLE.PTR)-1, NUM.COLUMNS) + 1
```

Row (1) is the top row in the table and column (1) is the leftmost column. If row headings were added to the table then column (1) refers to the row headings. If the table contains column headings, row (1) corresponds to the headings.

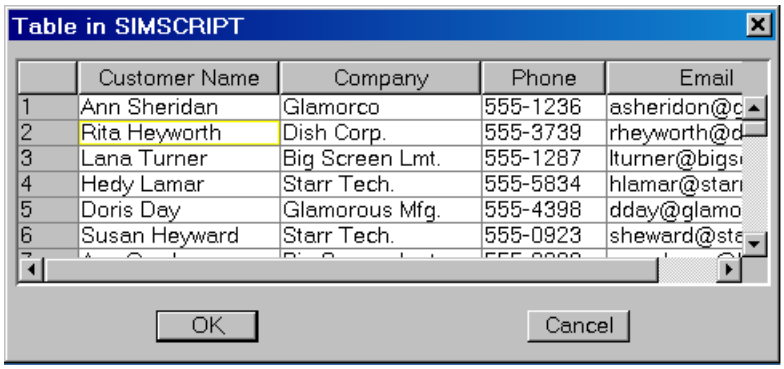

Table 5-3: Typical use of a SIMSCRIPT Table

Example 5.9: Using a 2d table in a dialog box

```
Main
''  Display a dialog containing a table.  To create in SIMSCRIPT STUDIO:
'' 1) Create a dialog called "my_dialog_box.frm" (see Example 5.1)
'' 2) Click on the table tool button on the right palette
'' 3) Position the table by dragging it with mouse.
'' 4) Double-click on the table to display properties.
'' 5) Change Field name to "my_table_field"
'' 6) Change number of rows to 10, number of columns to 2
'' 7) Check the "show column headings" box
'' 8) Click OK, Build the project

Define DIALOG.PTR, TABLE.PTR as pointer variables
Define TABLE.VALUES as a 1-dim text array
Define CELL.ROW, CELL.COLUMN as integer variables
Define .NUM.ROWS to mean 11
Define .NUM.COLUMNS to mean 2

Show DIALOG.PTR with "my_dialog_box.frm"
Let TABLE.PTR = DFIELD.F("my_table_field", DIALOG.PTR)

Reserve TABLE.VALUES(*) as .NUM.ROWS*.NUM.COLUMNS

'' label each cell in the table by its own row and column
For CELL.ROW = 1 to .NUM.ROWS do
  For CELL.COLUMN = 1 to .NUM.COLUMNS do
    Let TABLE.VALUES((CELL.ROW-1) * .NUM.COLUMNS + CELL.COLUMN) =
     CONCAT.F(ITOT.F(CELL.COLUMN), ",", ITOT.F(CELL.ROW))
  Loop
Loop

'' initialize the fields
Let DARY.A(TABLE.PTR) = TABLE.VALUES(*)

'' display the dialog box
If ACCEPT.F(DIALOG.PTR,0) eq "OK"
  Let SELECTED.ROW = DIV.F(INT.F(DDVAL.A(TABLE.PTR))-1, .NUM.COLUMNS) + 1
  Let SELECTED.COLUMN = MOD.F(DDVAL.A(TABLE.PTR)-1, .NUM.COLUMNS) + 1

  List SELECTED.ROW, SELECTED.COLUMN
Always

'' wait for user to press return
```

```
Read as /

End
```

5.6.10 Text Box

A Text box is a commonly used field type which allows you to place a single line of editable text into your dialog. Text boxes have the following properties:

- **Label** – The text appearing on the left-hand side of the box.
- **Width** – Size of the input box in font units. (average number of characters it can contain).
- **Text** – The text string initially shown in the box.
- **Selectable Using Return** – If true, the control routine will be called when user presses the **Return** key after typing text.

Use the **DTVAL.A**. attribute to initialize and retrieve the text. If you wish for the control routine to be called when the user presses the <return> key, set the **Selectable using Return** property in SIMSCRIPT Studio. Some code to initialize and retrieve text from a dialog is:

```
Let DTVAL.A(DFIELD.F("my_text_field", DIALOG.PTR)) = "Hello world"
Let TEXT.VALUE = DTVAL.A(DFIELD.F("my_text_field", DIALOG.PTR))
```

Example 5.10: Using a text box in a dialog

```
Main
''  Display a dialog containing text box.  To create in SIMSCRIPT STUDIO:
'' 1) Create a dialog named "my_dialog_box.frm" (see Example 5.1)
'' 2) Click on the text box tool button on the right palette
'' 3) Position the text box by dragging it with mouse
'' 4) Double-click on the value box to display properties.
'' 5) Change Field name to "my_text_box_field"
'' 6) Click OK, Build the project

Define DIALOG.PTR as a pointer variable

Show DIALOG.PTR with "my_dialog_box.frm"
Let DTVAL.A(DFIELD.F("my_text_field", DIALOG.PTR)) = "Hello world"

If ACCEPT.F(DIALOG.PTR, 0) eq "OK"
  Write DTVAL.A(DFIELD.F("my_text_field", DIALOG.PTR)) as
   "User entered: ", T*, /
  Read as  /
Always

End
```

5.6.11 Tree View List

Lists of items can be viewed hierarchically with items containing other items. Each item consists of a label and an optional jpeg image, and each item can have list of sub items associated with it. By clicking an item, the user can expand and collapse the associated list of sub items.

Using SIMSCRIPT Studio you can specify the initial set of items and sub items in the tree. Using the "Tree Properties" dialog, click on the "Add" button to add the top-most items. To add sub items, click on any item the click on the Add button. You can change the label, icon, and field name by clicking on an item, setting these fields, then clicking on the "Update" button.

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

```
<top_item_name>/<sub_item_name>/<sub_sub_item_name>/. . .
```

It is not necessary to include separate text values for the parent items.

For example, if you wanted to show following items:
a. "Reno" contained in "Nevada" which is in "USA"
b. "Las Vegas" contained in "Nevada" which is in "USA"
c. "Berlin" contained in "Germany" which is in "Europe".
The **DARY.A** attributes should contain the following text strings:

```
Define TREE.ITEMS as a 1-dim text array
Reserve TREE.ITEMS(*) as 3

Let TREE.ITEMS(1) = "USA/Nevada/Reno"
Let TREE.ITEMS(2) = "USA/Nevada/Las Vegas"
Let TREE.ITEMS(3) = "Europe/Germany/Berlin"

Let DARY.A(DFIELD.F("my_tree_field", DIALOG.PTR) = TREE.ITEMS(*)
```

To specify a jpeg picture to place next to the item name, use the "|" character after the name specification to separate the item name from jpeg file (or resource) name (without extension). To show an image next to a container item, add that item to the list. In the above example, suppose we want to:

a. show the bitmap "US_flag.jpg" next to the "USA" item.
b. show the jpeg file "Poker_chip.jpg" next to the "USA/Nevada/Las Vegas" item.
c. show the jpeg "German_flag.jpg" next to "Europe/Germany":

```
Reserve TREE.ITEMS(*) as 4

Let TREE.ITEMS(1) = "USA|US_flag"
Let TREE.ITEMS(2) = "USA/Nevada/Las Vegas|Poker_chip"
Let TREE.ITEMS(3) = "Europe/Germany|German_flag"
Let TREE.ITEMS(4) = "Europe/Germany/Berlin"
```

If you need to use the "|" or "/" characters in your item name, you can make them literal using a preceding 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 "2". You can set the selected item in the tree by setting **DDVAL.A** and redisplaying the field. From above, to set the selected item to "Berlin":

```
Let DDVAL.A(DFIELD.F("tree_field", DIALOG.PTR)) = 4
Display DFIELD.F("tree_field", DIALOG.PTR)
```

The control routine will be called whenever any item in the list is selected, but not when an item is expanded or collapsed. From the control routine, your program can inspect the **DDVAL.A** attribute to get the index of the selected item.

```
Let TREE.ITEMS(*) = DARY.A(DFIELD.F("my_tree_field", DIALOG.PTR))
Write TREE.ITEMS(DDVAL.A(DFIELD.F("my_tree_field", DIALOG.PTR)) as
 "Item selected is ", T *, /
```

Example 5.11: Show a tree control in a dialog box

```
Main
''  Display a dialog with a value box.  To create in SIMSCRIPT STUDIO:
'' 1) Create a dialog called  "my_dialog_box.frm" (see Example 5.1)
'' 2) Click on the tree tool button on the right palette
'' 3) Position the tree by dragging it with mouse
'' 4) Double-click on the tree to display properties.
'' 5) Change Field name to "my_tree_field"
'' 6) Click OK, Build the project

Define DIALOG.PTR as a pointer variable
Define TREE.ITEMS as a 1-dim text array

Show DIALOG.PTR with "my_dialog_box.frm"

Reserve TREE.ITEMS(*) as 4
Let TREE.ITEMS(1) = "USA|US_flag"
Let TREE.ITEMS(2) = "USA/Nevada/Reno|Poker_chip"
Let TREE.ITEMS(3) = "USA/Nevada/Las Vegas|Poker_chip"
Let TREE.ITEMS(4) = "Europe/Germany/Berlin|German_flag"

'' Initialize the dialog box tree field. Set the items and the initially
'' selected item
Let DARY.A(DFIELD.F("my_tree_field", DIALOG.PTR) = TREE.ITEMS(*)
Let DDVAL.A(DFIELD.F("my_tree_field", DIALOG.PTR)) = 1

If ACCEPT.F(DIALOG.PTR, 0) eq "OK"
   Write TREE.ITEMS(DDVAL.A(DFIELD.F("my_tree_field", DIALOG.PTR)) as
     "Item selected is ", T *, /
  Read as /
Always

End
```

Table 5-4: Dialog box containing a tree.

5.6.12 Value Box

A value box is used to receive or show a single numerical value to the user. A lower and upper bound can be specified for a value box input field. The width of the value box is dependant on its Minimum and Maximum. Use SIMSCRIPT Studio to set the following properties:

- **Label** – The title appearing on the left-hand side of the box.
- **Min, Max** – The range of values 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.
- **Precision** – Precision is used to format output and round input. It defines the number of digits to the right of the decimal point. Negative values can be used to round to 10s, 100s, etc. (0 = integer value, 1 = 0.1, 2 = 0.01, -1 = rounded to 10's, -2 = rounded to 100's)
- **Value** – The initial value displayed in the 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 callback routine will be notified when the user presses the **Return** key the control has input focus.

Your SIMSCRIPT program can initialize or retrieve the contents using the **DDVAL.A** attribute.

```
Let DDVAL.A(DFIELD.F("value_field", DIALOG.PTR)) = 12.5
Let VALUE.REAL = DDVAL.A(DFIELD.F("value_field", DIALOG.PTR))
```

Example 5.12: Using a value box

```
Main
''  Display a dialog with a value box.  To create in SIMSCRIPT STUDIO:
'' 1) Create a dialog called  "my_dialog_box.frm" (see Example 5.1)
'' 2) Click on the value box tool button on the right palette
'' 3) Position the value box by dragging it with mouse
'' 4) Double-click on the value box to display properties.
```

```
'' 5) Change Field name to "value_field"
'' 6) Change the Precision to "1"
'' 7) Click OK, Build the project

Define DIALOG.PTR as a pointer variable

Show DIALOG.PTR with "my_dialog_box.frm"
Let DDVAL.A(DFIELD.F("value_field", DIALOG.PTR)) = 12.5

If ACCEPT.F(DIALOG.PTR, 0) eq "OK"
  Write DDVAL.A(DFIELD.F("value_field", DIALOG.PTR)) as
   "User entered: ", D(4,1), /
 Read as /
Always

End
```

5.7 Tabbed Dialogs

Tabbed dialog boxes contain overlapping pages of fields. They allow you to create complex dialog boxes without using much screen space. Related fields can be grouped into the same page. The user can expose a page by clicking on its heading on top of the page.

Using SIMSCRIPT Studio, you can create a tabbed dialog by dragging one or more tab pages from the Studio's palette. Add fields to tab pages by first selecting the desired page, then dragging the field onto it. You can drag fields out of a page to place them back into the dialog box. Each page in the tab control has its own set of properties. Double click on a page's heading to edit the following properties:

- **Label** – The text label shown on the heading of the page.
- **Icon Name** – The resource or file name (without extension) of the jpeg file shown next to the page's label. Use the "Browse Resources" button to select from one of the built in JPEG images, or click on the "Browse Files" button to locate a JPEG file.
- **Verify before hide** –All value boxes in this page must be validated before the page can be hidden by the user. In other words, if any value is out of range, another page will not be exposed until the fields have been corrected by the user.

You do not need to make any changes to your source code to use tabbed dialog boxes. SIMSCRIPT treats the controls in separate tab pages as if they all belong directly to the dialog box. You should therefore make sure that controls in different pages do not share the same field name.

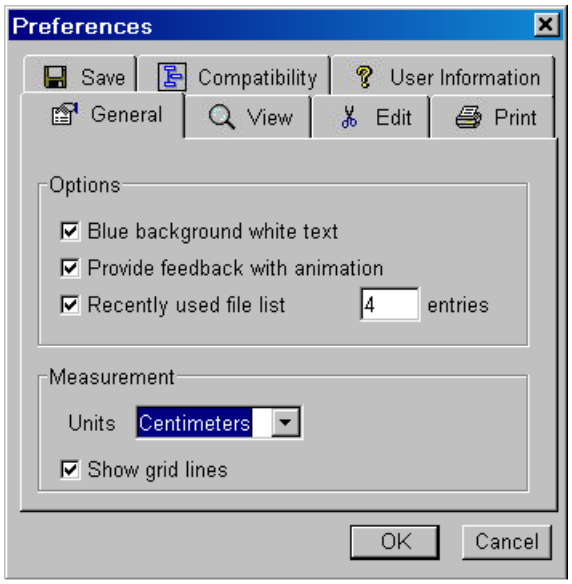

Table 5-5:Tabbed dialog box created by SIMSCRIPT Studio

5.8 Dialog box Properties

Dialog boxes have various properties that allow you to specify the title, initial placement on the screen, tab key traversal ordering, and how the dialog should be handled by ACCEPT.F.

Using SIMSCRIPT Studio you can change the properties of the dialog box itself by double-clicking on the background of the dialog editor's window. Dialog properties include the following:

- **Library Name** – Your program should use this name in the SIMSCRIPT "show" statement when loading the dialog box.
- **Modal Interaction** – If checked, your program will not return from ACCEPT.F until a button is clicked on that has the "terminating" property set. While this dialog is displayed, the user will not be able to click on any other windows in the application.
- **Title** – Text displayed in the top or the dialog frame.
- **Multiple rows of tab page headers** – If your dialog contains tab pages, setting this flag will cause page headers to appear in multiple rows when there are too many to display in the page area. Otherwise the user will be able to scroll page headers into view.
- **Position** – Use these items to define where on the screen the dialog box will appear when shown. Using the radio-box, you can specify which corner of the dialog will be offset from the lower left-hand corner of the screen. For example, if **Top 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 top right corner of the dialog box. This distance is specified in "screen coordinates" where the width and height of the computer screen are each 100 units.

• **Tab key traversal** – The ordering of items in the list defines the order in which to transfer input focus from one control to the next when the user presses the **Tab** key. Use the up- and down-arrow keys to shift the tab ordering of controls.
• **Action taken by ACCEPT.F** – You can also define the modeless behavior of **ACCEPT.F** when the "Modal interaction" flag is cleared. Select one of the following radio buttons:

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.

5.8.1 Dialog positioning in SIMSCRIPT

The initial location of the dialog box can be set programmatically using the **LOCATION.A** attribute of the form pointer. As far as your program is concerned, The lower left-hand corner of the screen maps to coordinate (0,0) while the upper right-hand corner maps to (32767,32767).

Using the DDVAL.A attribute of the dialog box pointer, you can specify at runtime which corner of the dialog box to position. Acceptable values for DDVAL.A are as follows:

0 → Position ignored. Place the dialog box in the center of the application's window.
1 → Bottom left corner
2 → Bottom right corner
3 → Top left corner
4 → Top right corner

For example, suppose we want to show the dialog box in the top right corner of the screen.

```
Let DDVAL.A(DIALOG.PTR) = 4
Let LOCATION.A(DIALOG.PTR) = LOCATION.F(32767,32767)
```

The value of **LOCATION.A** is updated automatically whenever the user repositions the dialog. The allows you to save the location of the dialog before your program quits, then redisplay it in the same place.

```
Write LOCATION.X(DIALOG.PTR), LOCATION.Y(DIALOG.PTR) as
 "Dialog box is at ", D(6,0), ", ", D(6,0), /
```

5.9 Predefined Dialog Boxes

Dialog boxes are designed to be general enough to receive a wide variety of data type from the user. However, some dialog boxes are commonly used in many different applications for a specific purpose. Toolkits provide standard dialog boxes for message display, file browsing and font selection. The SIMSCRIPT runtime library provides an interface for these built-in dialogs.

5.9.1 Simple Message Box

If you wish to *only* display a message and not present the user with a decision, the easiest way is to call the **`MESSAGEBOX.R`** routine. Pass the message text and the title of the dialog to this routine and a dialog is displayed containing a button labeled "OK" or "Continue". **`MESSAGEBOX.R`** will not return until the user presses the button.

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

5.9.2 Custom Message Box

In some cases you will want only to display a simple message to the user and allow him to answer "yes", "no", or "cancel". This necessity is so common that most toolkits provide built in dialog boxes just for that purpose. SIMSCRIPT allows you to add *message boxes* to your program to make use of these dialogs.

Table 5-6: SIMSCRIPT message box.

More advanced message boxes can be created in SIMSCRIPT Studio and are loaded and displayed by like regular dialogs. Your program can display **Alert, Question, Information** and **Stop** dialog boxes. To add a message box to your project, right-click on "graphics.sg2" then select "new" from the popup. Select "Simple message box" from the drop down list and click on the "Create" button. The "Message Box Editor" can then be

used to edit these dialogs. Here, a message dialog can be set up to have one of the following five styles:

a. Plain
b. Stop Sign
c. Question
d. Exclamation
e. Information

Each style shows a different icon in its window. It can 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.

Any one of these buttons can be designated as the *default* button. This button is activated when the user presses the <return> key.

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 SIMSCRIPT Studio or by your program. The **DARY.A** attribute of the dialog points to a text array containing the message.

The field name returned by **ACCEPT.F** indicates 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:

Example 5.13: Using the two varieties of message box.

```
''  Display a message box.  To create in SIMSCRIPT STUDIO:
'' 1) Right click on "graphics.sg2" then select "new"
'' 2) From the "Create" dialog select "Simple message box" from the
''    drop down list. Click on the "Create" button.
'' 3) From the "Message Box Properties" dialog enter
''    "my_message_box.frm" as the Name.
'' 4) Check the radio buttons "Question", "Yes, No, Cancel" and
''    under "Default button" check "Yes".
'' 5) Click on the OK button, then build the project
Main
Define MESSAGE.PTR as a pointer variable
Define TEXT.LINES as a 1-dim text array
Reserve TEXT.LINES(*) as 2
```

```
Show MESSAGE.PTR with "my_message_box.frm"
Let TEXT.LINES(1) = "Do you want to save changes to the project"
Let TEXT.LINES(2) = "'My_Project.dat'?"
Let DARY.A(MESSAGE.PTR) = TEXT.LINES(*)
Select case ACCEPT.F(MESSAGE.PTR, 0)
Case "YES"     Call MESSAGEBOX.R("User selects YES.", "Response")
Case "NO"      Call MESSAGEBOX.R("User selects NO.", "Response")
Case "CANCEL"  Call MESSAGEBOX.R("User selects Cancel.", "Response")
Endselect
End
```

5.9.3 File Browser 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:

```
Define FILTER, TITLE, PATH.NAME, FILE.NAME as text variables
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 respectfully. The **TITLE** parameter is shown in the title bar of the dialog.

5.9.4 Font Browser Dialog

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:

```
Define TITLE, FAMILY.NAME as text variables
Define POINT.SIZE, ITALIC.DEGREE, BOLDFACE.DEGREE
      as integer variables
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** (see Chapter 3). **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 **""**.

6. Menu Bars

In a typical application, the user interacts with a menu bar attached to the top of the window. For most applications, the entire range of functionality is accessible through the menu bar. Given its widespread use, most users expect to be able to control the program through the 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-pane to be displayed. Clicking on an item inside a menu causes it to be selected. Cascadeable menus are supported, meaning that menus can contain other menus and so on.

Like other forms, a menubar is constructed using SIMSCRIPT Studio, and displayed in your program using the **`ACCEPT.F`** function. As with dialogs and palettes, the program can employ a control routine to receive notification of menu item selection while a simulation is running.

6.1 Constructing a Menu Bar in SIMSCRIPT Studio

Using SIMSCRIPT Studio you can add a menu bar to your project by right clicking on the graphics library and selecting the "New" item from the popup menu. Select "Menu bar" from the list and click OK. To edit an existing menu bar, double-click its name shown in the contents pane under "graphics.sg2".

The menu bar editor will show a representative of your menu bar near the top of the edit window. To the right of the edit window a vertical toolbar contains three buttons used to add both menus and menu items. To add a menu, click on the second button from the top of the toolbar, then click in the menu bar to deposit the menu. At this time, a placeholder is displayed under the menu which allows you to add menu items. This is done by clicking on third button from the top of the tool bar, then inside the placeholder. Clicking on any menu will show or hide its placeholder contains its items. The menu-bar editor is shown in Figure 6.1:

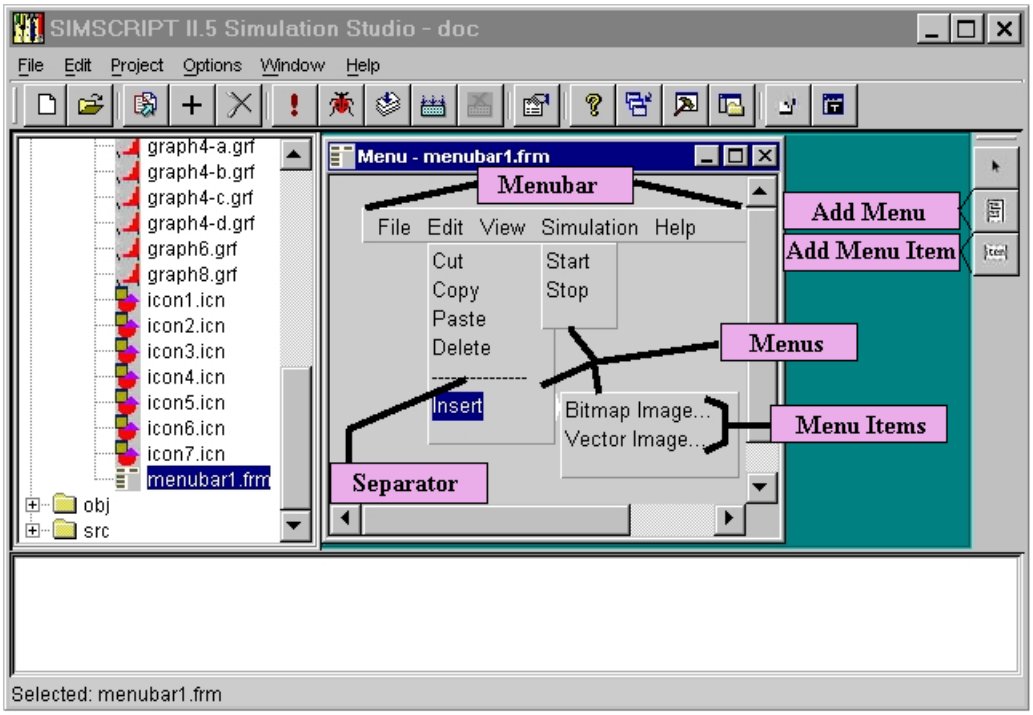

Figure 6-1: Editing a menu-bar in SIMSCRIPT Studio

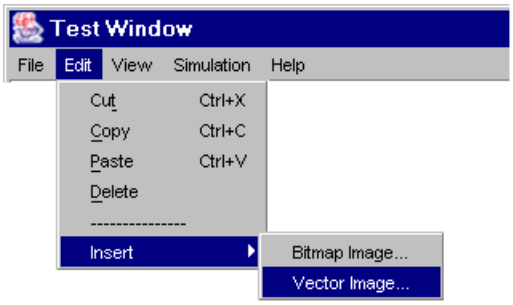

Figure 6-2: The menu-bar from Figure 6.1 displayed in a program.

You can double-click on the menu bar, menus, or menu items shown in the edit window to display a properties dialog. This allows you to specify (among other things) the text of the menu or item, as well is its field name which is used in your program. These dialog boxes contain the following:

6.1.1 Menu Bar Properties

• **Library name** – The name of the object in the graphics.sg2 library. This name should be passed to the SIMSCRIPT **`SHOW`** statement.
• **Action taken by ACCEPT.F** – You can also define how **`ACCEPT.F`** will behave when displaying the form in your program. **`ACCEPT.F`** 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 this process will be resumed. If there is no simulation running and hence no active process, the **Synchronous** mode is used automatically.
2. **Synchronous**: Regardless of the simulation, **ACCEPT.F** will not return until a status value of "1" is returned from the control routine.
3. **Don't wait**: **ACCEPT.F** will not wait for any action by the user but will return immediately leaving the menu bar visible. Subsequent selection of items in the menus will invoke the control routine.

6.1.2 Menu Properties

• **Field name** – Any menu added to the menu bar or another menu can be accessed from inside an application by specifying a **Field** name. (Usually the program will be interested in the field name of the menu item, not the menu).
• **Label** – The text identifying the menu which appears on the menu bar.
• **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.

6.1.3 Menu Item Properties

• **Field name** – Any menu item can be accessed from inside an application by specifying its **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.

6.2 Showing the Menu Bar in your program

SIMSCRIPT provides some easy to use constructs which will allow you to load and display your menu bar. The SIMSCRIPT "show" will create a new instance of the menu bar and assign it to a pointer variable, but not yet make it visible.

```
Show MENUBAR.PTR with "my_menu_bar.frm"
```

The text in quotes is the name of the menu-bar that was assigned in SIMSCRIPT Studio when the menu bar was first created. The text must match up exactly with the name provided in the menu-bar editor, or a runtime error will be generated. Display the menu-bar using the **ACCEPT.F** function as follows:

```
Let FIELD.ID = ACCEPT.F(MENUBAR.PTR, 'MENUBAR.RTN')
```

Pass to **ACCEPT.F** the pointer to the menu-bar display entity used in the SHOW statement. A control routine should be provided as the second argument (described below). **ACCEPT.F** will normally not return until the control routine indicates termination by setting its yielded STATUS argument to '1' instead of zero.

6.3 Writing a Control Routine for the Menu bar

Normally, you will want the menu bar to remain visible while your program is executing other code. A separate routine should be written for the menubar called a *control routine*. When the user selects a menu item while the program is running, the SIMSCRIPT runtime library will call the control routine passing the field name of the menu item selected. Code in the control routine can match the field name against the known menu item fields, taking whatever action is necessary when a match is found.

A control routine heading looks like this:

```
Routine MENUBAR.RTN given FIELD.NAME, MENUBAR.PTR
```

```
      yielding FIELD.STATUS
```

The FIELD.NAME text argument will contain the field name specified in SIMSCRIPT Studio. The MENUBAR.PTR argument is the pointer to the menu-bar display entity. Your control routine will fill in the FIELD.STATUS integer argument with one of the following values:

0 -- Continue to display the menubar.
1 -- Stop displaying the menubar. Return from **ACCEPT.F**

Example 6.1:

```
'' - Use control routine to receive callbacks from a menu bar
'' To create in SIMSCRIPT STUDIO:
'' 1) Right click on "graphics.sg2" then select "new" from popup
'' 2) Select "Menu bar" in the list, click on the "Create" button
'' 3) Click on the Menu icon (right toolbar) then on the menubar
''    in the edit window to add a menu.
'' 4) Double click on the text of the new menu to show properties.
''    Set its Label to "Menu" and field name to "MENU".
'' 5) Click on the menu to show its pane.  Add four items
''    by clicking on the Menu Item toolbar button (third from top)
''    then in the small pane.
'' 5) Double click on items to set text and properties.
''    Set item field names and labels to "FIRST", "SECOND, "THIRD"
''    and "QUIT".
'' 6) Build the project.
Routine MENUBAR.RTN given FIELD.NAME, MENUBAR.PTR yielding FIELD.STATUS
define FIELD.NAME as text variable       '' name of field
define MENUBAR.PTR as pointer variable    '' pointer to bar
define FIELD.STATUS as an integer variable '' set to 0,1
Select case FIELD.NAME
  Case "FIRST"   Write as "First was selected" , /
  Case "SECOND"  Write as "Second was selected", /
  Case "THIRD"   Write as "Third was selected", /
  Case "QUIT"    let FIELD.STATUS = 1 '' 1 => return from ACCEPT.F
  Default
Endselect
Return
End

Main
Define MENUBAR.PTR as a pointer variable
Define FIELD.ID as a text variable
Show MENUBAR.PTR with "menubar1.frm"  '' load the dialog box
Let FIELD.ID = ACCEPT.F(MENUBAR.PTR, 'MENUBAR.RTN')    '' display the dialog
box
End
```

6.4 Using a Menu Bar within a Simulation

If the menu bar needs to be available to the user while a simulation is running, you will not want the program to “disappear” into the ACCEPT.F function. To prevent this, the menu bar must be “asynchronous”. This means that the simulation will not stop running when a call is made to ACCEPT.F—instead the current process is suspended, allowing the simulation to continue.

There are two steps to setting up this asynchronous interaction. First, the menu bar should be defined as "Asynchronous". From SIMSCRIPT Studio double click on the menu-bar shown in the edit window to bring up the "Menu Bar Properties" dialog box described above. Then click on the radio button labeled "Asynchronous".

The next step is to add to your code a simple process in to show your menu bar. The call to **ACCEPT.F** should be made inside this process. When **ACCEPT.F** is called, the process will be suspended, therefore you should not try to use this same process to perform any other simulation related activities. The process should look something like this:

```
Process MENU.BAR
Define MENUBAR.PTR as a pointer variable
Define FIELD.ID as a text variable
Show MENUBAR.PTR with "menubar.frm"
Let FIELD.ID = ACCEPT.F(MENUBAR.PTR, 'MENUBAR.RTN')
End
```

Example 6.2: Menu bar displayed during a simulation

In this example a menu bar can be used while a simulation is running. Every second the process "TEST" will print a message. While this is happening, the user can select items in the menus without interrupting the simulation.

```
'' To create in SIMSCRIPT Studio:
'' 1) Right click on "graphics.sg2", select new from the popup menu
'' 2) Double click on the container shown in the Menu bar editor.
'' 3) Set library name to "menubar2.frm". Select the "Asynchronous"
''    radio button. Click OK.
'' 4) Create the contents of the menu bar by following the steps
''    3-6 outlined in Example 6.1
Preamble
Processes include MENUBAR, TEST
End

Routine MENUBAR.RTN given FIELD.ID, FORM yielding STATUS
Define FORM as a pointer variable
Define FIELD.ID as a text variable
Define STATUS as an integer variable
Write FIELD.ID as "Button selected: ", T *, /
If FIELD.ID eq "QUIT"
  Let STATUS = 1
Always
End

Process MENUBAR
Define MENUBAR.PTR as a pointer variable
Define FIELD.ID as a text variable
Show MENUBAR.PTR with "menubar2.frm"
Let FIELD.ID = ACCEPT.F(MENUBAR.PTR, 'MENUBAR.RTN')
Stop
End

Process TEST
While 1 eq 1 Do
  Wait 1 unit
  Write as "Message from test process", /
Loop
End
```

```
Main
Let TIMESCALE.V = 100  '' 1 sec per unit
Activate a TEST now
Activate a MENUBAR now
Start simulation
End
```

6.5 Changing Menus, Sub-Menus and Menu Items at Runtime

Occasionally, items in the menu bar will may need to be changed at runtime. For example, deactivating or "graying-out" of items is needed when function indicated by the item is no longer possible to perform. For functions that toggle off and on, a check mark can be placed next to the item. A SIMSCRIPT program can access menus and menu items the same way controls in a dialog box are accessed, through the DFIELD.F function.

6.5.1 Accessing Menus and Menu Items

The **DFIELD.F** function can be used to obtain a pointer to a display entity representing the menu or item. To get a pointer to a menu, pass the menu's field name along with the menu bar pointer to the **DFIELD.F** function. To get a pointer to the menu item, pass a pointer to its menu and its field name to **DFIELD.F**.

```
Define MENU.PTR, MENU.ITEM.PTR as pointer variables
Let MENU.PTR = DFIELD.F(MENU.FIELD.NAME, MENUBAR.PTR)
Let MENU.ITEM.PTR = DFIELD.F(ITEM.FIELD.NAME, MENU.PTR)
```

The **DFIELD.F** call will recursively search through all menus and sub-menus for the item specified. Therefore you can usually get a pointer to the menu item using only one call to **DFIELD.F** with the menu bar pointer as its parameter:

```
Let MENU.ITEM.PTR = DFIELD.F(ITEM.FIELD.NAME, MENUBAR.PTR)
```

The **DARY.A** (text array) attribute of the menu pointer contains the labels of the items shown in the menu. If for some reason you need to know the ordinal position of the menu item that was last selected by the user, it is available through the **DDVAL.A** attribute of the menu containing the selected item. The indexing begins at 1.

```
Define ITEM.ARRAY as a 1-dim text array
Define ITEM.NUM as an integer variable
Define SELECTED.ITEM.TEXT as a text variable
. . .
Let ITEM.NUM = DDVAL.A(MENU.PTR)
Let ITEM.ARRAY(*) = DARY.A(MENU.PTR)
Let SELECTED.ITEM.TEXT = ITEM.ARRAY(ITEM.NUM)
```

6.5.2 Accessing Menus in Menus

Menu bars can be *cascadeable,* i.e. menus can contain other menus. The relationship between menus and sub-menus is specified in SIMSCRIPT Studio by dragging a menu onto another menu. 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 field 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. There is no limit to the number or layers of menus.

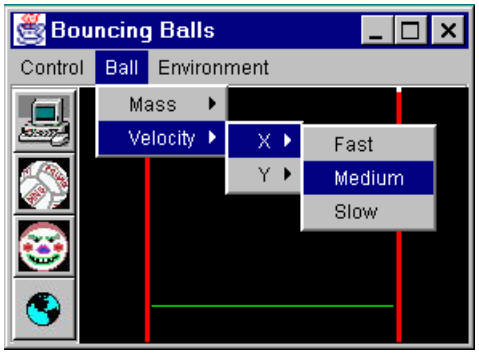

Figure 6-3: Cascading menus used in a small simulation program.

6.5.3 Adding Checkmarks to Menu Items

Your program can dynamically set and clear check marks next to any menu item. To display the check mark, set the **DDVAL.A** attribute of the menu item field pointer to "1". Clear the mark by setting the attribute to "0". Before the check mark is actually drawn or erased from the menu, the item must be re-displayed using the SIMSCRIPT **display** statement.

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)
```

6.5.4 Deactivate Menu Items

A menu item can be deactivated (grayed out) by calling the **SET.ACTIVATION.R** routine. The arguments are a pointer to the menu item entity, and the integer value "0" to deactivate the item, and "1" to activate it.

```
'' Deactivate the item with field name "RUN"
Call SET.ACTIVATION.R(DFIELD.F("RUN", MENUBAR.PTR), 0)

'' Activate the item with field name "RUN"
Call SET.ACTIVATION.R(DFIELD.F("RUN", MENUBAR.PTR), 1)
```

Another way to active and deactivate items is by concatenating the special code ASCII 129 to the beginning of the menu item text.

```
Let ITEM.ARRAY(*) =
      DARY.A(DFIELD.F("MENU.ITEM.FIELD", MENUBAR.PTR))

'' Activate the second item by concatenating 129
Let ITEM.ARRAY(2) = CONCAT.F(ATOT.F(129), ITEM.ARRAY(2))

'' Deactivate the second item by concatenating 130
Let ITEM.ARRAY(2) = CONCAT.F(ATOT.F(130), ITEM.ARRAY(2))
```

Example 6.3: Check, uncheck, activate and deactivate menu items.
This example shows how individual menu items can be grayed out or have a check mark placed next to them.

```
'' To create in SIMSCRIPT STUDIO:
'' 1) Right click on "graphics.sg2" then select "new" from popup
'' 2) Select "Menu bar" in the list, click on the "Create" button.
''    Set the name of the menu bar to "menubar3.frm".
'' 3) Click on the Menu icon (right toolbar) then on the menubar
''    in the edit window to add a menu.
'' 4) Double click on the text of the new menu to show properties.
''    Set its Label to "Menu" and field name to "MENU".
'' 5) Click on the menu to show its pane.  Add five items
''    by clicking on the Menu Item toolbar button (third from top)
''    then in the small pane.
'' 5) Double click on items to set text and properties.
''    Set item field names and labels to "ITEM", "ACTIVATE,
''    "DEACTIVATE", "CHECK", "UNCHECK".
'' 6) Build the project.
Routine MENUBAR.RTN given FIELD.NAME, MENUBAR.PTR yielding FIELD.STATUS
define FIELD.NAME as text variable       '' name of field
define MENUBAR.PTR as pointer variable    '' pointer to bar
define FIELD.STATUS as an integer variable '' set to 0,1
define MENUITEM.PTR as a pointer variable
Let MENUITEM.PTR = DFIELD.F("ITEM", MENUBAR.PTR)
Select case FIELD.NAME
  Case "ACTIVATE"
    Call SET.ACTIVATION.R(MENUITEM.PTR, 1)
  Case "DEACTIVATE"
    Call SET.ACTIVATION.R(MENUITEM.PTR, 0)
  Case "CHECK"
    Let DDVAL.A(MENUITEM.PTR) = 1
    Display MENUITEM.PTR
  Case "UNCHECK"
    Let DDVAL.A(MENUITEM.PTR) = 0
    Display MENUITEM.PTR
  Default
Endselect
Return
End

Main
Define MENUBAR.PTR as a pointer variable
Define FIELD.ID as a text variable
Show MENUBAR.PTR with "menubar3.frm"  '' load the dialog box
```

```
Let FIELD.ID = ACCEPT.F(MENUBAR.PTR, 'MENUBAR.RTN') '' display form
End
```

Example 6.4: Menus inside menus

In this example cascading menus are constructed.

```
'' To create in SIMSCRIPT STUDIO:
'' 1) Right click on "graphics.sg2" then select "new" from popup
'' 2) Select "Menu bar" in the list, click on the "Create" button
'' 3) Click on the Menu icon (right toolbar) then on the menubar
''    in the edit window to add a menu.
'' 4) Double click on the text of the new menu to show properties.
''    Set its Label to "Menu" and field name to "MENU".
'' 5) Click on the menu to show its pane.  Add a menu to the menu
''    by clicking on the Menu toolbar button then in the menu pane
''    Set the field name and Label of the new menu to SUB-MENU
'' 6) Repeat step 5 adding a new menu to the sub menu.  Label it
''    SUB-SUB-MENU. Add a menu item labelled SUB-SUB-ITEM to the sub
''    sub menu.
'' 7) Build the project.
Routine MENUBAR.RTN given FIELD.NAME, MENUBAR.PTR yielding FIELD.STATUS
Define FIELD.NAME as text variable       '' name of field
Define MENUBAR.PTR as pointer variable    '' pointer to bar
Define FIELD.STATUS as an integer variable '' set to 0,1
Write FIELD.NAME as T *, " was selected", /
Return
End

Main
Define MENUBAR.PTR as a pointer variable
Define FIELD.ID as a text variable
Show MENUBAR.PTR with "menubar4.frm"  '' load the dialog box
Let FIELD.ID = ACCEPT.F(MENUBAR.PTR, 'MENUBAR.RTN')    '' display the dialog
box
End
```

6.6 Popup Menus

Popup menus are not contained inside the menu bar, but instead are displayed when the user clicks with the right mouse button inside a SIMSCRIPT graphics window. They are sometimes referred to as *context menus* because a different menu can easily be displayed depending on both where and when the user right-clicks in the window.

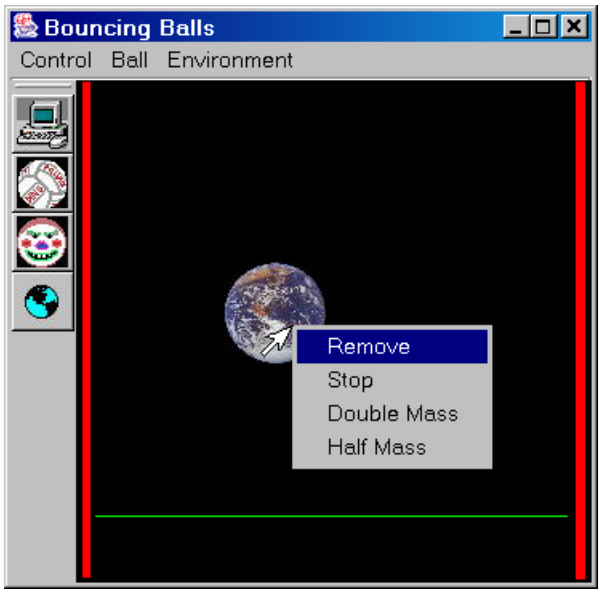

Figure 6-4: Right clicking on the “earth” icon shows a special popup menu.

6.6.1 Creating and Displaying the Popup Menu

Popup menus are not created in SIMSCRIPT Studio, but are specified in your program. A popup menu is displayed using the POPUP.F function. An array of text values to be shown in the menu is passed to the function. It then displays the menu at the mouse click location. POPUP.F does not return until the user has made a menu selection. The text of the selected item is returned. The user can decline to make a selection by clicking outside the menu. In this case POPUP.F will return the empty string “”.

Here is an example of some code that displays a popup menu containing the items “Cut”, “Copy” and “Detail”:

```
Define POPUP.ITEMS as a 1-dim text array
Reserve POPUP.ITEMS(*) as 3
Let POPUP.ITEMS(1) = "Cut"
Let POPUP.ITEMS(2) = "Copy"
Let POPUP.ITEMS(3) = "Detail"
Select case POPUP.F(POPUP.ITEMS(*))
  case "Cut"          . . .
  case "Copy"         . . .
  case "Detail"       . . .
  default
Endselect
```

6.6.2 Using Popup Menus in a Simulation

As in the case with menu bars, it is desirable for the simulation to continue to run while waiting for the user to right-click in the window. When the right-click comes, the

simulation should be interrupted in order to allow the program to display the popup menu and process the selection.

A popup menu can behave asynchronously by creating a separate process for interacting with it. This process should use the READLOC.R routine given locator style 16 to suspend itself until the mouse is clicked. At this time the process calls POPUP.F to display the popup menu.

After READLOC.R returns, the global variable G.4 will contain the segment id of the icon that was last clicked on. You can compare G.4 with the SEGID.A attribute of known icon display entities to decide which popup menu to show.

Example 6.5: Showing a popup menu in a simulation
In this example a popup menu can be used while a simulation is running. Every second the process "TEST" will print a message. While this is happening, the user can click in the window with the right mouse button to display the popup.

```
Preamble
Processes include POPUP.MENU, TEST
Define POPUP.F as a text function
End

Process POPUP.MENU
Define POPUP.ITEMS as a 1-dim text array
Define X, Y as double variables
Define XFORM as an integer variable
Reserve POPUP.ITEMS(*) as 3
Let POPUP.ITEMS(1) = "Red"
Let POPUP.ITEMS(2) = "Green"
Let POPUP.ITEMS(3) = "Blue"
While 1 eq 1 do
  Call READLOC.R given 0, 0, 16 yielding NEWX, NEWY, XFORM
  Select case POPUP.F(POPUP.ITEMS(*))
    case "Red"   write as "Red item selected", /
    case "Green" write as "Green item selected", /
    case "Blue"  write as "Blue item selected", /
    default      write as "Nothing selected", /
  Endselect
Loop
Release POPUP.ITEMS(*)
End

Process TEST
While 1 eq 1 Do
  Wait 1 unit
  Write as "Message from test process", /
Loop
End

Main
Let TIMESCALE.V = 100  '' 1 sec per unit
Activate a TEST now
Activate a POPUP.MENU now
Start simulation
End
```

7. Palettes

Toolbars and palettes are also popular components in a user interface. (For the sake of SIMSCRIPT Studio, we will refer to either component as a *palette*.) A palette is basically a horizontal or vertical bar containing a row (or column) of buttons, with each button showing a little picture icon. Your application may need a palette at the top of the window to provide quick access to commonly used items. Many applications also allow users to drag various items from palettes on the side into the adjacent window.

In SIMSCRIPT, a palette can be attached to any edge of the window. The buttons contained on a palette can be by typical push buttons, or can toggle (stay down when pressed.) The user can drag with the mouse the image on a palette button and drop it into the canvas portion of the window. In this case your program is notified of the action allowing you to create a display entity representing the object that was dragged.

Palettes are designed using the SIMSCRIPT Studio *Palette Editor*. A runtime, your program code will load and display the palette using the SHOW or DISPLAY statements.

7.1 Constructing a Palette in SIMSCRIPT Studio

Adding a palette to you program is fairly simple. From SIMSCRIPT Studio, right click on “graphics.sg2” and select “new” from the popup menu. From the “Create New Graphic” dialog box, select “Palette” in the list. Provide a name for your palette. As a convention, palette names usually have the extension “.frm”. This name will be provided to your program’s SHOW statement. Clicking on the “Create” button will show your new palette in a palette editor window.

The palette editor will show a vertical toolbar attached to the right side of SIMSCRIPT Studio containing three buttons. To add a new palette button to your SIMSCRIPT palette first click on the second button from the top of this toolbar. Then position the mouse over the container shown in the palette editor and click down once again.

The order of the buttons can be rearranged by clicking and dragging a button to the desired position in the container. Delete a button by selecting it with the mouse and pressing the <Delete> key.

To see what your palette will look like when the application is run by using the “test window”. Display your palette in a test window by clicking on the rightmost button in the toolbar at the top of the SIMSCRIPT Studio window.

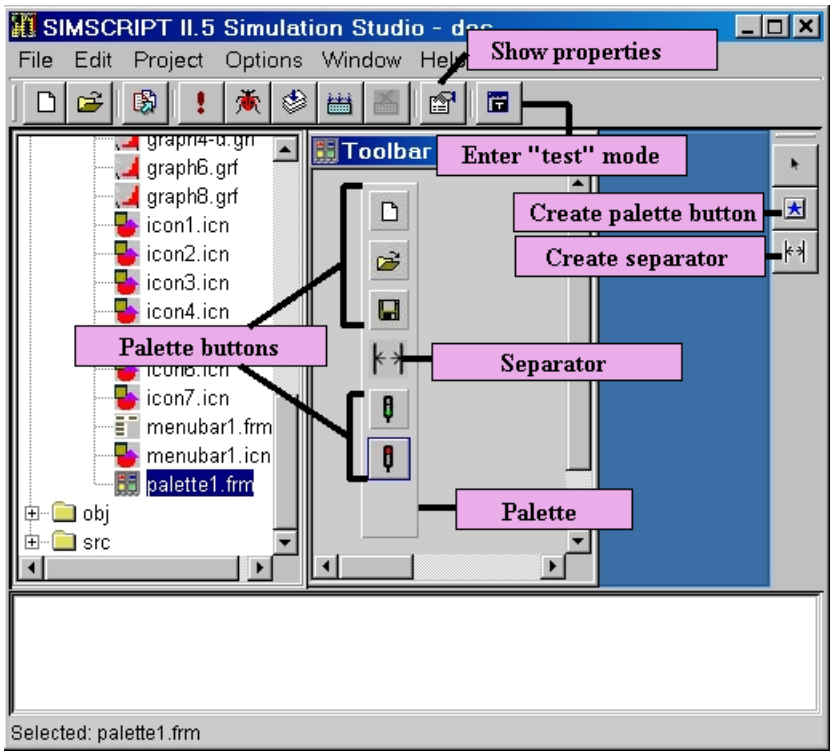

Figure 7-1: Palette Editor in SIMSCRIPT Studio

7.1.1 Properties of the Palette

Palettes can be attached to any edge of the application window, or be floating (not unlike a modeless dialog box.) Palettes can also be *dockable* meaning they can be moved by the user from one edge of the window to another while running the application. Palettes cannot be resized; they are always sized to fit their contents.

The "Palette Properties" dialog box can be viewed by double-clicking on the palette button container shown in the edit window. This dialog box contains the following:

• **Library Name** – The name of this palette in the graphics library which should be provided to the SHOW statement.
• **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.
• **Action taken by ACCEPT . F**– Specify one of the following *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, 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.
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.

7.1.2 Properties of a Palette Button

You will need to double-click on each button in the palette to specify its properties. At the top of the "Palette Button Properties" dialog is the "Field name" box. Specify a unique text value that will be used in by the program code to gain access to the data associated with this button.

• **Field Name** – Any button added to the palette can be accessed from inside an application by specifying a **Field** name. The field name is passed to the callback routine whenever the button is clicked on.
• **Icon Name** – The name of the JPEG file (without extension) icon displayed on the face of the palette button. Pressing either the "Browse resources" or "Browse files" button to set the name.
• **Status Message** – You can specify a "Status Message" that will appear at run-time in the status bar at the bottom of the window whenever the user passes the mouse over the button.
• **Tool Tip** – Specify text to be presented in a small popup window when the user stops the mouse over the button (without clicking on it).
• **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. The DDVAL.A attribute of the button field pointer will be "1" if the button is down, or "0" otherwise.
3. **Draggable** – Allows the user to hold the mouse button down and drag an outline of the palette button into the canvas portion of the adjacent 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. For **Color Buttons** the button will be colored using the values specified in the "Red", "Green" and "Blue" boxes.
• **Button Face Color (Red,Green,Blue)** – You can set the color of the **Color Button**s using these value boxes. Color is defined by the percentage of Red, Green, and Blue (range [0-100]).
• **Browse resources** – To use a built-in icon, click on the "Browse resources" button. A dialog will be shown which allows you to select from a picture list of all predefined images.

• **Browse files** – Brings up a file browser dialog that will allow a JPEG file to be selected. (NOTE: Always select a JPEG file in the same directory as your executable).

7.1.3 Specifying a Button Face Image

To specify the JPEG icon shown on the face of a button, first make sure that the "Icon Button" radio button is selected. From this point you have a choice of using one of the "built-in" icons provided by SIMSCRIPT, or you can specify the name of a jpeg file that will be shown on the face of your palette button.

If you wish to use an icon defined in a JPEG file, it is recommended that you FIRST copy the file into the same directory as your executable and "graphics.sg2" file (usually the "executable" sub-directory under your current project directory). These JPEG files must remain with the executable whenever your program is run. Click on the "Browse files" button to choose a JPEG file for the button face.

7.1.4 Palette Separators

It may be desirable to show some buttons in the palette in groups. This is common when the buttons are functionally distinct from the rest. The palette editor allows separators to be placed in between two buttons.

You can add a separation by dragging a separator item from the toolbar on the right

7.2 Showing the Palette in your Program

The SIMSCRIPT "show" will create a new instance of a palette and assign it to a pointer variable.

```
Show PALETTE.PTR with "my_palette.frm"
```

The text in quotes is the name of the menu-bar that was assigned in SIMSCRIPT Studio. The text must match up exactly with the name provided in the menu-bar editor, or a runtime error will be generated. Display the menu-bar using the ACCEPT.F function as follows:

```
Let FIELD.ID = ACCEPT.F(PALETTE.PTR, 'PALETTE.RTN')
```

7.3 Writing Code for a Palette

The program needs to be notified of user interaction with the palette. You can provide a *control routine* to ACCEPT.F which will be called whenever the user clicks on a button in the palette, or drags a button image into the window. The SIMSCRIPT runtime library

automatically calls your control routine as the simulation runs (asynchronously) without the need to poll.

7.3.1 Writing a Control Routine for a Palette

The first step is to define your control routine. Passed to this routine will be a pointer to the PALETTE form and the field id (assigned in SIMSCRIPT Studio) of the palette button that was clicked on by the user. The format of this routine is as follows:

```
Routine PALETTE.CTL given FIELD.ID, FORM yielding STATUS
```

The routine should contain a "select case" statement which compares the FIELD.ID to each known palette button "field name" and performs the appropriate action when a match is found. The STATUS variable should be set to '0' if you want to continue displaying the palette. If this variable is set to –1, the palette will be erased, and the suspended process used to display the palette will be activated.

Here is an example of a typical control routine:

```
Routine PALETTE.CTL given FIELD.ID, FORM yielding STATUS
Define FORM as a pointer variable
Define FIELD.ID as a text variable
select case FIELD.ID
  case "SHOW_STATISTICS"
    call SHOW.STATISTICS
  case "RESET_VALUES"
    call RESET.VALUES
  default
endselect
let STATUS = 0  '' never return from accept.f
return
```

Example 7.1: Simple palette with two buttons
In this example a palette is displayed which contains two buttons that can be clicked on. The field name of the selected button is printed to the screen. The control routine is called first with the field name "INITIALIZE" and when a click outside the palette is detected, it is called with the field name "BACKGROUND".

```
'' To create in SIMSCRIPT Studio:
'' 1) Right click on "graphics.sg2", select new from the popup menu
'' 2) Double click on the container shown in the Palette editor.
'' 3) Set library name to "Palette1.frm". Select the "Synchronous"
''    radio button. Click OK.
'' 4) Add two palette buttons using the toolbar on the right then clicking
''    inside the container in the edit window.
'' 5) Double click on the top button.  Select "Icon button" in the dialog
''    Set Icon name to START_L.
'' 6) Repeat step 5 for the second button but set Icon name to STOP_L.
Routine PALETTE.CTL given FIELD.ID, FORM yielding STATUS
Define FORM as a pointer variable
Define FIELD.ID as a text variable
Write FIELD.ID as "Button selected: ", T *, /
End
```

```
Main
Define PALETTE.PTR as a pointer variable
Define FIELD.ID as a text variable
Show PALETTE.PTR with "palette1.frm"
Let FIELD.ID = ACCEPT.F(PALETTE.PTR, 'PALETTE.CTL')
End
```

7.3.2 Writing a Process for an Asynchronous Palette

If the palette is to be used while a simulation is running, you will want to define it as "asynchronous" within the "Palette Properties" dialog box described above. This means that the simulation will not stop running when a call is made to ACCEPT.F—instead the current process is suspended, allowing the simulation to continue.

There are two steps to setting up this asynchronous interaction. First, create a simple process to show your palette. The call to ACCEPT.F should be made inside this process. When ACCEPT.F is called, the process will suspended, therefore you should not try to use this same process to perform any other simulation related activities. The process should look something like this:

```
Process PALETTE
Define PALETTE.PTR as a pointer variable
Define FIELD.ID as a text variable
Show PALETTE.PTR with "palette.frm"
Let FIELD.ID = ACCEPT.F(PALETTE.PTR, 'PALETTE.CTL')
End
```

Example 7.2: Palette displayed during a simulation
In this example a palette can be clicked on while a simulation is running. Every second the process "TEST" will print a message. While the is happening, the user can click on palette buttons without interruption of the simulation.

```
'' To create in SIMSCRIPT Studio:
'' 1) Right click on "graphics.sg2", select new from the popup menu
'' 2) Double click on the container shown in the Palette editor.
'' 3) Set library name to "Palette2.frm". Select the "Asynchronous"
''    radio button. Click OK.
'' 4) Add two palette buttons using the toolbar on the right then clicking
''    inside the container in the edit window.
'' 5) Double click on the top button.  Select "Icon button" in the dialog
''    Set Icon name to START_L.
'' 6) Repeat step 5 for the second button but set Icon name to STOP_L.
Preamble
Processes include PALETTE, TEST
End

Routine PALETTE.CTL given FIELD.ID, FORM yielding STATUS
Define FORM as a pointer variable
Define FIELD.ID as a text variable
Write FIELD.ID as "Button selected: ", T *, /
End

Process PALETTE
Define PALETTE.PTR as a pointer variable
Define FIELD.ID as a text variable
Show PALETTE.PTR with "palette2.frm"
```

```
Let FIELD.ID = ACCEPT.F(PALETTE.PTR, 'PALETTE.CTL')
End

Process TEST
While 1 eq 1 Do
  Wait 1 unit
  Write as "Message from test process", /
Loop
End

Main
Let TIMESCALE.V = 100  '' 1 sec per unit
Activate a TEST now
Activate a PALETTE now
Start simulation
End
```

7.3.3 Handling Toggle Palette Buttons

As was mentioned in Chapter 7.1.2, 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 in a down (=1) or up (=0) state.

You can set the initial state of the button before calling **ACCEPT.F** but after using the **SHOW** statement. In the following code, we set the palette button with field name "TOGGLE_FIELD" to be initially *down*.

```
…
Show PALETTE.PTR with "my_palette.frm"
Let DDVAL.A(DFIELD.F("TOGGLE_FIELD", PALETTE.PTR)) = 1
```

Your control routine is called after the user changes the state of the button. Here the program code acquires the current state of the button by examining **DDVAL.A**.

```
…
Let BUTTON.DOWN = DDVAL.A(DFIELD.F(("TOGGLE_FIELD", PALETTE.PTR))
```

7.3.4 Handling Drag and Drop Palette Buttons

Defining a palette button as being "Dragable" in SIMSCRIPT Studio allows the user to click down on the button, then drag its outline to the canvas of the window. When the mouse button is released, the palette's control routine is called.

In some cases, you will need to know the exact location in the window that the mouse was released. To get this coordinate value, set the global variable **DINPUT.V** to a display entity pointer. When the user releases the mouse, the drop point can be retrieved through

the **LOCATION.A** attribute of **DINPUT.V**. The **DIVAL.A** attribute of **DINPUT.V** will contain the viewing transform number corresponding to that drop coordinate.

Example 7.3: Dragging items from a palette

In this example a palette can be clicked on while a simulation is running. Any one of three buttons on the palette can be dragged and dropped into the canvas of the window. An icon representing the item is displayed at the drop location.

```
'' To create in SIMSCRIPT Studio:
'' 1) Right click on "graphics.sg2", select new from the popup menu
'' 2) Double click on the container shown in the Palette editor.
'' 3) Set library name to "Palette3.frm". Select the "Asynchronous",
''    radio button. Click OK.
'' 4) Add three palette buttons using the toolbar on the right then clicking
''    inside the container in the edit window.
'' 5) Double click on each button.  Mark the "Icon button" and "Draggable"
''    fields. For each button, set "Icon name" fields to "QUEUE_L",
''    "ROUTER_L" and "CLOUD_L".  For each button set "Field name" to
''    "QUEUE", "ROUTER", and "CLOUD".
'' 6) Create three icons named "queue.icn", "router.icn" and "cloud.icn".
''    For each icon use the "Edit/Insert JPeg" option to add a bitmap.
''    Name the jpeg images "QUEUE_L", "ROUTER_L", and "CLOUD_L"
''    Respectfully.  Use the "Edit/Icon Properties" menu to ensure each
''    icon has the "Automatic recenter" box checked.
Preamble
Processes include PALETTE, DUMMY
Graphic entities include LOCATOR
End

Routine PALETTE.CTL given FIELD.ID, FORM yielding STATUS
Define FORM as a pointer variable
Define FIELD.ID as a text variable
Define ICON.PTR as a pointer variable
Define X,Y as real variables
Let X = LOCATION.X(DINPUT.V)   Let Y = LOCATION.Y(DINPUT.V)
Select case FIELD.ID
  case "QUEUE"
    Display ICON.PTR with "queue.icn" at (X,Y)
  case "ROUTER"
    Display ICON.PTR with "router.icn" at (X,Y)
  case "CLOUD"
    Display ICON.PTR with "cloud.icn" at (X,Y)
  default
Endselect
End

Process PALETTE
Define PALETTE.PTR as a pointer variable
Define FIELD.ID as a text variable
Show PALETTE.PTR with "palette3.frm"
Let FIELD.ID = ACCEPT.F(PALETTE.PTR, 'PALETTE.CTL')
End

Process DUMMY
While 1 eq 1 Do
  Wait 1 unit
Loop
End

Main
```

```
Create a LOCATOR called DINPUT.V
Activate a DUMMY now
Activate a PALETTE now
Start simulation
End
```

8. Windows

SIMSCRIPT allows the programmer to create 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 SIMSCRIPT runtime library to your program whenever the user manipulates the window, (i.e. resizing, closing, moving the thumb on a scroll bar, etc.).

A window is created by calling the **OPENWINDOW.R** routine described below:

```
Routine OPENWINDOW.R given XLO, XHI, YLO, YHI, TITLE,
   MAPPING.MODE yielding WINDOW.PTR
```

Is not always necessary to call **OPENWINDOW.R** to create your window. If the program attempts to display graphics without a window, one is created automatically. To create a window by program code, call **OPENWINDOW.R** *before* displaying any graphics.

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 the range `0..32767`. The point `(0,0)` defines the lower left corner of the screen, and `(32767,32767)` is located in the upper right 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. Assume the world coordinate system defined by the SETWORLD.R routine is (**WORLD.XLO WORLD.XHI WORLD.YLO WORLD.YHI).** The following modes are available:

MAPPING.MODE = 0: Contents mapped to largest centered square within window.

MAPPING.MODE = 1: WORLD.XLO 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 a window to display your icons, graphs, and forms is accomplished by associating the window with one or more viewing transformations. In this way the **VXFORM.V** variable not only specifies which viewing transformation will be used to draw subsequent graphics, but also identifies the window to contain the graphics. Objects drawn under the same **VXFORM.V** value *cannot* appear in two different windows.

Viewing transformations are assigned to a window using the **SETWINDOW.R**. Set **VXFORM.V** to the desired transformation number, and then call **SETWINDOW.R** given the **WINDOW.ID** of the window to contain the objects drawn under that transformation.

Example 8.1: Creating two windows:
In this example the OPENWINDOW.R routine is used to display two separate windows, each containing an icon.

```
Main
Define WINDOW1.PTR, WINDOW2.PTR as pointer variables
Define ICON1.PTR, ICON2.PTR as pointer variables
''-- 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 transformation 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
Let VXFORM.V = 2   '' prepare to display icon1
Display ICON1.PTR with "icon1.icn" at (16383,16383)
Let VXFORM.V = 3   '' prepare to display icon2
Display ICON2.PTR with "icon2.icn" at (16383,16383)
While 1 eq 1 do
    call HANDLE.EVENTS.R(1)
Loop
End
```

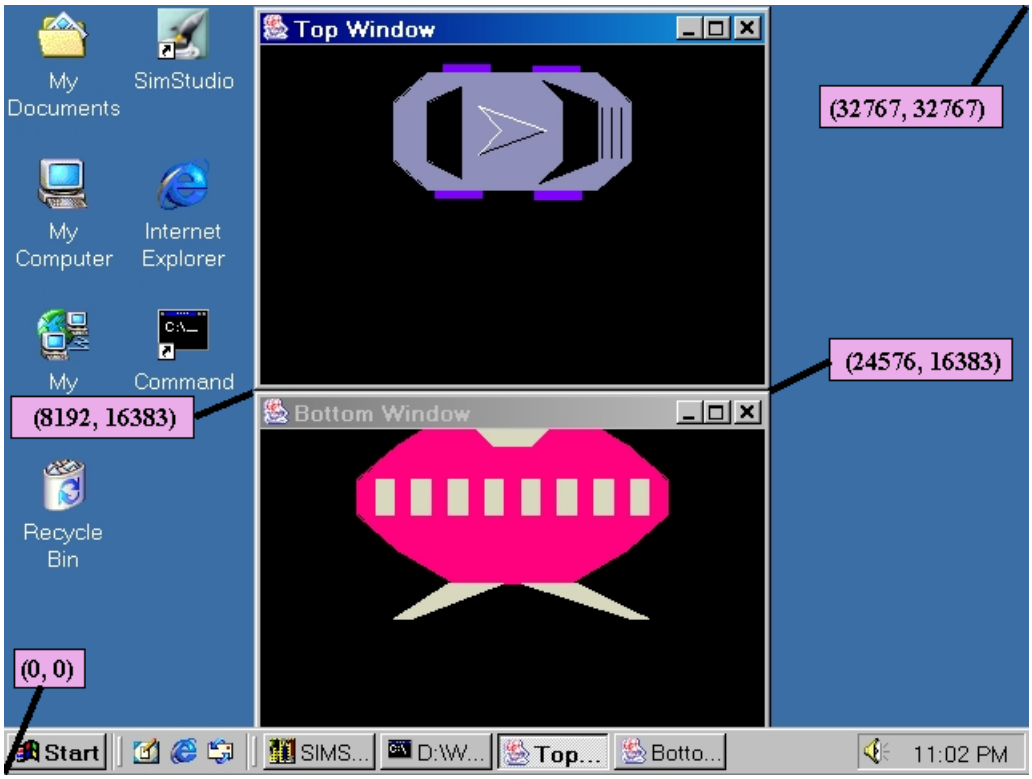

Figure 8-1: Two windows opened by SIMSCRIPT.

The coordinate space of the computer screen ranges from (0,0) in the lower left corner to (32767,32767) in the upper right.

8.1 Setting and Getting the Attributes and Events of a Window

Calling **`OPENWINDOW.R`** yields a display entity pointer. The **`DFIELD.F`** function can then be used on this display entity to access window *fields* that are predefined by the SIMSCRIPT runtime library

8.1.1 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.

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))
```

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	DDVAL.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 Y 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.

8.2 Window Events

Whenever the user resized, moves scrolls or closes a window, the program will sometimes need to take some sort of action. For example, if the user moves a scroll bar, you may want to "pan" the contents of the window. SIMSCRIPT will generate asynchronous callbacks as a result of any action performed on the window, eliminating the need to continuously "poll" the window fields for changes. A *control routine* can be defined by the programmer to handle these events. Window control routines work the same way as dialog box, menu bar, and palette control routines do. The control routine is assigned to the window by calling **SET.WINCONTROL.R**.

```
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
```

Table 8.2 lists all events that can be received by the control routine. The "Default Action" column explains what the SIMSCRIPT runtime library will do after the control routine is called if the yielded argument **BLOCK.DEFAULT** is set to "0". The "Affected Fields" are defined in Table 8.1 and set by the runtime library before the control routine is called.

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.
MOUSECLICK	None	XCLICK, YCLICK, BUTTONDOWN, BUTTON, DOUBLECLICK	Sent whenever any mouse button is pressed down, or lifted up.
MOUSEMOVE	None	XMOVE, YMOVE	Sent whenever mouse movement occurs.

Example 8.2: Using a control routine to receive window events
This example illustrates how your program can receive notification of changes made to the window by the user at runtime. A window control routine receives three of the predefined event types: CLOSE, MOUSEMOVE and MOUSECLICK.

```
Routine WINDOW.CONTROL 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
Select case EVENT.NAME
Case "CLOSE"
    Write as "Attempt to close window... ", /
    Let BLOCK.DEFAULT = 1    ''do not terminate here!
Case "MOUSEMOVE"
    Write DDVAL.A(DFIELD.F("XMOVE", WINDOW.PTR)),
          DDVAL.A(DFIELD.F("YMOVE", WINDOW.PTR)) as
      "Mouse moved to ", D(7,1), ", ", D(7,1), /
Case "MOUSECLICK"
    Stop     '' terminate on click in window
Default
Endselect
End

Main
Define WIN.PTR as a pointer variable
Call OPENWINDOW.R given 16383, 32768, 8192, 24576,
    "Example 2 Window", 0 yielding WIN.PTR
Call SETWINDOW.R(WIN.PTR)
Call SET.WINCONTROL.R(WIN.PTR, 'WINDOW.CONTROL')
While 1 eq 1 Do
    Call HANDLE.EVENTS.R(1)
Loop
End
```

8.3 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 common after *zooming* into a rectangular section of your graphics area. To create the scrollable window, set the HSCROLLABLE and / or VSCROLLABLE fields of the window pointer before displaying the window. For example:

```
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 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, suppose the total area occupied by the graphics is defined by (`t.xlo, t.xhi, t.ylo, t.yhi`). Suppose also that you only want the user to see a portion of this space and have therefore provided the (smaller) visible coordinate space to the

SETWORLD.R routine as (w.xlo, w.xhi, w.ylo, w.yhi). In this case the thumb sizes should be 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))
```

Movement 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 is set to distance from the left side of the horizontal scroll thumb to the left 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.

Example 8.3: Pan and Zoom

Simple pan and zoom can be implemented using the SETWORLD.R routine. Attaching scroll bars to your window not only provide a way to pan, but also indicate to the user which portion of the scene is currently being viewed. In this example, the user can zoom in by clicking with the left mouse button, zoom out with the right button, and pan using the scroll bars

```
Preamble
Define t.xlo, t.xhi, t.ylo, t.yhi as real variables
Define Z.XLO, Z.XHI, Z.YLO, Z.YHI as real variables
Define WINDOW.PTR as a pointer variable
End

'' this routine pans the display by shifting the world to the
'' given location
Routine PAN given XLO, YHI
  Define XLO, YHI as real variables
  Define WIDTH, HEIGHT as real variables

  '' Compute new dimensions for zoom rect
  Let WIDTH = Z.XHI-Z.XLO
  Let HEIGHT = Z.YHI-Z.YLO
  Let Z.XLO = XLO
  Let Z.XHI = XLO + WIDTH
  Let Z.YHI = YHI
  Let Z.YLO = YHI - HEIGHT

  '' pan by changing the world coordinate system
  Call SETWORLD.R(Z.XLO, Z.XHI, Z.YLO, Z.YHI)
End

'' taking the center point for the zoom, this routine will
'' adjust the world coordinates via SETWORLD.R and then update
'' the size of the scroll bar thumb accordingly
Routine ZOOM given CLICK.X, CLICK.Y, FACTOR
  Define CLICK.X, CLICK.Y, FACTOR as real variables
  Define CENTER.X, CENTER.Y as real variables
  Define WIDTH, HEIGHT as real variables
```

```
  '' convert NDC mouse click coordinates to zoom coordinates
  Let CENTER.X = (Z.XHI - Z.XLO) * CLICK.X / 32768.0 + Z.XLO
  Let CENTER.Y = (Z.YHI - Z.YLO) * CLICK.Y / 32768.0 + Z.YLO

  '' Compute new dimensions for zoom
  Let WIDTH = MIN.F((Z.XHI-Z.XLO) / FACTOR, t.xhi-t.xlo)
  Let HEIGHT = MIN.F((Z.YHI-Z.YLO) / FACTOR, t.yhi-t.ylo)

  '' Limit zoom to world boundaries
  Let CENTER.X = MIN.F(MAX.F(CENTER.X, WIDTH/2.0 - T.XLO),
    T.XHI - WIDTH/2.0)
  Let CENTER.Y = MIN.F(MAX.F(CENTER.Y, HEIGHT/2.0 - T.YLO),
    T.YHI-HEIGHT/2.0)

  '' compute new zoom rectangle using center and size values
  Let Z.XLO = CENTER.X - WIDTH / 2.0
  Let Z.XHI = CENTER.X + WIDTH / 2.0
  Let Z.YLO = CENTER.Y - HEIGHT / 2.0
  Let Z.YHI = CENTER.Y + HEIGHT / 2.0

  '' perform zoom by setting world to zoom rectangle
  Call SETWORLD.R(Z.XLO, Z.XHI, Z.YLO, Z.YHI)

  '' Update scroll bars
  Let DDVAL.A(DFIELD.F("HTHUMBSIZE", WINDOW.PTR)) = WIDTH /
    (t.xhi-t.xlo)
  Let DDVAL.A(DFIELD.F("VTHUMBSIZE", WINDOW.PTR)) = HEIGHT /
    (t.yhi-t.ylo)
  Let DDVAL.A(DFIELD.F("HTHUMBPOS", WINDOW.PTR)) = Z.XLO /
    (t.xhi-t.xlo)
  Let DDVAL.A(DFIELD.F("VTHUMBPOS", WINDOW.PTR)) =
     (T.YHI-Z.YHI) / (t.yhi-t.ylo)

  Display DFIELD.F("HTHUMBSIZE", WINDOW.PTR)
  Display DFIELD.F("VTHUMBSIZE", WINDOW.PTR)
End

'' This routine is called by the SIMSCRIPT runtime when
'' the user clicks in the window or somehow positions
'' the scrollbar thumb.
Routine WINDOW.CONTROL 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
Select case EVENT.NAME
Case "HSCROLL"   '' Pan horiz or vert
  write as "User moved horiz scroll bar", /
  Call PAN(t.xlo + (t.xhi-t.xlo) *
      DDVAL.A(DFIELD.F("HTHUMBPOS", WINDOW.PTR)), Z.YHI)

Case "VSCROLL"
  write as "User moved vert scroll bar", /
  Call PAN(Z.XLO, t.yhi - (t.yhi-t.ylo) *
      DDVAL.A(DFIELD.F("VTHUMBPOS", WINDOW.PTR)))

Case "MOUSECLICK"
  If DDVAL.A(DFIELD.F("BUTTONDOWN", WINDOW.PTR)) gt 0
    If DDVAL.A(DFIELD.F("BUTTON", WINDOW.PTR)) gt 0
      Call ZOOM(DDVAL.A(DFIELD.F("XCLICK", WINDOW.PTR)),
                DDVAL.A(DFIELD.F("YCLICK", WINDOW.PTR)), 0.5)
    Else
      Call ZOOM(DDVAL.A(DFIELD.F("XCLICK", WINDOW.PTR)),
```

```
                DDVAL.A(DFIELD.F("YCLICK", WINDOW.PTR)), 2.0)
    Always
  Always
Default
Endselect
End

Main
  Define ICON.PTR as a pointer variable
  Let Z.XLO = 0   Let Z.XHI = 32767
  Let Z.YLO = 0   Let Z.YHI = 32767
  Let T.XLO = 0   Let T.XHI = 32767
  Let T.YLO = 0   Let T.YHI = 32767
  Let VXFORM.V = 1
  Call OPENWINDOW.R (0, 26000, 2000, 30000, "Pan and Zoom Window", 0)
    yielding WINDOW.PTR
  Let DDVAL.A(DFIELD.F("HSCROLLABLE", WINDOW.PTR)) = 1
  Let DDVAL.A(DFIELD.F("VSCROLLABLE", WINDOW.PTR)) = 1
  Call SETWINDOW.R(WINDOW.PTR)
  Call SET.WINCONTROL.R(WINDOW.PTR, 'WINDOW.CONTROL')
  Display ICON.PTR with "window3.icn"
  Call ZOOM(Z.XHI / 2.0, Z.YHI / 2.0, 1.0)
  While 1 eq 1 Do
    Call HANDLE.EVENTS.R(1)
  Loop
End
```

8.4 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 panes of varying width; each containing text. You can define the width of each pane before the window is displayed, and set the text displayed in a pane after the window has been rendered.

To add a status bar to a window, the **PANEWIDTH** field of the window pointer must be assigned after the pointer is obtained from **OPENWINDOW.R**. Each element of the array in the **DARY.A** attribute of this field specifies the maximum number of characters visible in the corresponding pane.

```
Call OPENWINDOW.R given 16383, 32768, 8192, 24576,
    "Title", 0 yielding WIN.PTR
Reserve PANE.WIDTH(*) as 3
Let PANE.WIDTH(2) = 15
Let PANE.WIDTH(3) = 10
Let DARY.A(DFIELD.F("PANEWIDTH",WIN.PTR)) = PANE.WIDTH(*)
Call SETWINDOW.R(WIN.PTR)
```

Note that the width of the first status pane is determined automatically based on the width of the window. The width specification for the first status pane (i.e. `PANE.WIDTH(1)`) is always ignored.

Each element of the **DARY.A** attribute of the **STATUSTEXT** field defines the text to display in the corresponding pane. Usually this text will need to be updated as the

program runs. Do this by first obtaining a pointer to the **DARY.A** attribute of the **STATUSTEXT** field, changing the element, then re-displaying the field.

```
'' Specify initial text array to go in the status bar
Reserve STATUS.TEXT (*) as 3
Let STATUS.TEXT(1) = "One"
Let STATUS.TEXT(2) = "Two",
Let STATUS.TEXT(3) = "Three",
Let DARY.A(DFIELD.F("STATUSTEXT",WIN.PTR)) = STATUS.TEXT(*)

'' Update the status bar text in the middle of the program
Let STATUS.TEXT(*) = DARY.A(DFIELD.F("STATUSTEXT",WIN.PTR))
Let STATUS.TEXT(2) = "New status text"
Display DFIELD.F("PANEWIDTH",WIN.PTR)
```

The first pane in the status bar shows status text for a highlighted menu item or palette button. From the menu bar and palette editors in SIMSCRIPT Studio, this can be specified in the "Status message" box in the "Menu Item Properties", and "Palette Button Properties" dialogs. At runtime, existing text in this pane will be replaced as the mouse hovers over the menu item or palette button.

Example 8.4: Creating a window with a status bar:
In this example we show how the above window fields can be used to add a status bar to the bottom of a window. Both the size of each pane and its contents are controlled by the "PANEWIDTH" and "STATUSTEXT" fields.

```
Main
Define PANE.WIDTH as a 1-dim integer array
Define STATUS.TEXT as a 1-dim text array
Define WIN.PTR as a pointer variable
Define I as an integer variable
Reserve PANE.WIDTH(*) as 3
Reserve STATUS.TEXT(*) as 3
Let PANE.WIDTH(2) = 10
Let PANE.WIDTH(3) = 20
Let STATUS.TEXT(1) = "Pane one"
Let STATUS.TEXT(2) = "Pane two"
Let STATUS.TEXT(3) = "Count: 0"
Call OPENWINDOW.R given 16383, 32768, 8192, 24576,
    "Example 4 Window", 0 yielding WIN.PTR
Let DARY.A(DFIELD.F("PANEWIDTH",WIN.PTR)) = PANE.WIDTH(*)
Let DARY.A(DFIELD.F("STATUSTEXT",WIN.PTR)) = STATUS.TEXT(*)
Call SETWINDOW.R(WIN.PTR)
For I = 1 to 10000 do
  Let STATUS.TEXT(3) = CONCAT.F("Count: ", ITOT.F(I))
  Display DFIELD.F("STATUSTEXT", WIN.PTR)
  Call HANDLE.EVENTS.R(0)
Loop
End
```

9. Routines and Globals for SIMSCRIPT Graphics

The following is an exhaustive list of all graphics related routines and predefined global variables in SIMSCRIPT II.5:

Function `ACCEPT.F (FORM.PTR, CONTROL.RT)`
Arguments:

`FORM.PTR` A pointer to the graphic input form to be used. This pointer was obtained in the `SHOW` statement.

`CONTROL.RT` 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.

Purpose: Accept graphic input from the screen.

Notes: 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 2. Values are in real world coordinates. **`POINTS(1, ...)`** are the x-coordinates. **`POINTS(2, ...)`** are the corresponding y-coordinates.

Purpose: Draw a circle.

Notes: A circle is drawn, with the center at the first given point. The second given point is any point on the circumference. The circle is drawn with attributes set through **`FILLCOLOR.R`**, **`FILLSTYLE.R`**, and **`FILLINDEX.R`**.

Routine `CLEAR.SCREEN.R`
Purpose: Erases all graphics in the current screen.

Notes: No segments or entities are destroyed.

Attribute `CLOCK.A (DSPLENT)`
Mode: Double.

Argument:

DSPLENT Pointer to a GRAPHIC entity or to a DYNAMIC GRAPHIC entity.

Notes: Time of last position update. This value is maintained by the routine called through **MOTION.A**.

Routine CLOSE.SEG.R

Purpose: Close a segment.

Notes: 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. As a side effect, the value of **SEGID.V** is set to zero.

Routine CLOSEWINDOW.R (WINDOW.ID)

Arguments:

WINDOW.PTR Pointer. Identifier returned by **OPENWINDOW.R.**

Purpose: Closes a SIMSCRIPT window given its pointer.

Notes: Graphical entities contained in this window are NOT destroyed.

Attribute DARY.A (FIELD.PTR)

Mode: Array of text.

Argument:

FIELD.PTR A pointer to a field in a graphic input form.

Notes: Contains the lines of text 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)

Mode: Double.

Argument:

FIELD.PTR A pointer to a field in a graphic input form.

Purpose: Access the numeric value attribute of a field.

Notes: This is used to acquire or change information in one field of a form. For instance, in a value box it is the value which the user

entered or which was pre-set.

Routine DELETE.SEG.R (SEG.ID)

Arguments:

SEG.ID	Integer. Identifier of a segment, as produced by **OPEN.SEG.R**.

Purpose: Delete a segment.

Notes: 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	Text. The name of the field (assigned in SIMSCRIPT Studio).
FORM.PTR	A pointer to a graphic input form.

Purpose: Returns a pointer to the specified field.

Notes: 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.

Argument:

DSPLENT	Pointer to a GRAPHIC entity or to a DYNAMIC GRAPHIC entity.

Purpose: Associates a display routine with an instance of an entity.

Notes: 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)

Mode: Text

Argument:

FIELD.PTR	A pointer to a graphic input field.

Purpose: Access a text value associated with the field.

Notes: **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.

Purpose: 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.

Purpose: Draw a line or a polygon.

Notes: A filled area (polygon) is drawn connecting the indicated points. The area is drawn in the current fill color, index, and style specified through routines **FILLCOLOR.R, FILLINDEX.R**, and **FILLSTYLE.R**. The first and last points are automatically 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.

Purpose: Set color of subsequent fill areas.

Notes: Assign index values using the **GCOLOR.R** routine.

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

Purpose: Set style of subsequent fill hatch areas.

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`.

Purpose: Set style of subsequent fill areas.

Routine `FONTBOX.R(TITLE) yielding FAMILY.NAME, POINT.SIZE, BOLDFACE, ITALIC`

Arguments:

`TITLE`	String. The title for the font dialog box.
`FAMILY.NAME`	String. The name of the font selected in the dialog box.
`POINT.SIZE`	Integer. The size of the font selected in points.
`ITALIC`	Integer. Return value of the font slant selected by the user. The range is from 0 to 1000. For most fonts only two values are allowed.
`BOLDFACE`	Integer. Return value of the "boldness" of the font. The range is from 0 to 1000. For most fonts only two values are allowed.

Purpose: Show the system font dialog box.

Notes: Provides a predefined dialog box for font specification that can be brought up programmatically to allow the user to select system font attributes. The yielded arguments can be passed directly to **`TEXTSYSFONT.R.`**

Routine `GCOLOR.R (COLOR.INDEX, RR, GG, BB)`

Arguments:

`COLOR.INDEX`	Integer. Values range 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.

Purpose: Define a color index value.

Notes: Sets the color representation for subsequent use under the indicated color index. **RR,GG**, and **BB** are the portions of red, green, and blue to use. Index values are passed to routines like **FILLCOLOR.R, LINECOLOR.R, MARKCOLOR.R,** and **TEXTCOLOR.R.** If the color of an existing segment is redefined, it must be redrawn for the color change to be visible.

Routine GDEFERRAL.R (DEFER)

Arguments:

DEFER Integer.
1 = set deferral on
0 = set deferral off

Purpose: Set deferral status of entire system.

Notes: 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. A segment ID value as returned by **OPEN.SEG.R**.

DETECT Integer.
0 = set undetectable status
1 = set detectable status

Purpose: Make a segment detectable or not.

Notes: 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

Purpose: Set the highlighting status of a segment. A highlighted segment draws attention to itself on the display surface.

Notes: SIMSCRIPT graphics 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.

Purpose: Set or change the priority of a segment.

Notes: 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.

Routine `GUPDATE.R`

Purpose: Draws all un-segmented 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

Purpose: Make a segment visible or invisible.

Routine `HANDLE.EVENTS.R(WAIT.FOR.EVENT)`

Arguments:

`WAIT.FOR.EVENT`	Integer. 0—Return immediately after processing all immediate events. 1—Wait for a mouse or other event to occur.

Purpose: Call the event handler.

Notes: Handles toolkit events such as window resizing and mouse interaction. Necessary for tight loop constructs occupying a large amount of time.

Routine `LINEAR.R (DSPLYENT)`
Arguments:
`DSPLYENT` Pointer to a `DYNAMIC GRAPHIC` entity.

Purpose: Manage one object with linear motion.

Notes: 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. Values range from 0 to 255.

Purpose: Set color of subsequent line primitives.

Notes: Call before **`POLYLINE.R, SECTOR.R.`**

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

Purpose: Set style of subsequent line primitives.

Notes: Call before **`POLYLINE.R, SECTOR.R.`**

Routine `LINEWIDTH.R (WIDTH)`
Arguments:
`WIDTH` Integer. In NDC units (range 0 to 32767).

Purpose: Set width of subsequent lines.

Notes: Call before **`POLYLINE.R, SECTOR.R.`**

Routine `LISTBOX.SELECTED.R (LISTBOX.PTR, INDEX) Yielding SELECTED`
Arguments:

`LISTBOX.PTR`	Pointer to a list box `FIELD` within a form.
`INDEX`	Integer. Index into array of list items
`SELECTED`	Integer return value. `1` if item has been selected, `2` if it has been double-clicked on, `0` otherwise.

Purpose: Get selection status of an item in a list box.

Notes: Given a list box field pointer and an index into the array of items, 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.

Notes: 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)`
Argument:

`DSPLYENT`	Pointer to a graphic entity, dynamic or static.

Right hand side: 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.

Purpose: Provide location for modeling transformation.

Notes: 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)

Mode: 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**.

Arguments:

X	Real, in real world coordinates.
Y	Real, in real world coordinates.

Purpose: Set a present location given x and y.

Function LOCATION.X (DSPLYENT)

Mode: Real, in real world coordinates.

Arguments:

DSPLYENT	Pointer to a graphic entity, dynamic or static.

Purpose: Inquire the present X position.

Function LOCATION.Y (DSPLYENT)

Mode: Real, in real world coordinates.

Arguments:

DSPLYENT	Pointer to a graphic entity, dynamic or static.

Purpose: Inquire the present Y position.

Routine MARKCOLOR.R (COLOR.INDEX)

Arguments:

COLOR.INDEX	Integer. Ranges from 0 to 255.

Purpose: Set color of subsequent markers.

Notes: Use the **GCOLOR.R** routine to assign an index value. Call before **POLYMARK.R**

Routine MARKSIZE.R (SIZE)

Arguments:

SIZE	Integer. The value is 0 to 32767, in NDC units.

Purpose: 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

Purpose: 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.

Purpose: Display a dialog box containing a one-line message.

Notes: A modal dialog box containing one **OK** button the given message will be displayed. The routine returns when the user clicks on **OK**.

Attribute MOTION.A (DSPLYENT)
Mode: Subprogram variable. The subprogram does not return a value.

Argument: Pointer to a DYNAMIC GRAPHIC entity.

Purpose: Provides an animation velocity management routine.

Notes: 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.

Purpose: Set the scaling component of the system modelling transformation.

Notes: 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.

Purpose: Specifies translation (movement) component of a modeling transformation.

Notes: Call from within a display routine or before a call to **CLOSE.SEG.R**. The translation is cumulative with previous translations. All rotation specified through **MZROTATE.R** is performed before translation.

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.

Purpose: Reset the modeling transformation to that of the given object.

Notes: 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. Angle of rotation in radians. Positive values indicate counter-clockwise rotation.

Purpose: Set the rotation attribute of the modeling transformation.

Notes: 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

Purpose: Open a new segment.

Notes: A new graphic segment is opened and made able to accept graphic

primitive operations. The integer global variable **SEGID.V** is changed to the identifier of this new segment.

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.

Purpose: Opens a new SIMSCRIPT graphics window.

Notes: Opens up a SIMSCRIPT graphics window of the prescribed dimensions on the screen and returns a display entity for it. Using this routine, you can create more than one graphics window for your application. **SETWINDOW.R** can then be used to associate the current global viewing transformation number (**VXFORM.V**) 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 counter-clockwise rotation.

Subscript: Pointer to a GRAPHIC entity or to a DYNAMIC GRAPHIC entity.

Notes: Sets orientation of a graphic entity, for the modeling 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**.
Purpose:	Selection from a menu using the mouse.
Notes:	Waits for the user to select a graphic entity with the mouse. If an entity is not selected, **INDEX** is set to zero. Otherwise, yields the index of the highest priority entity that was clicked on.

Routine POLYLINE.R (COUNT, POINTS(*))

Arguments:

COUNT	Integer. Number of points in the line.
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.
Purpose:	Draw a multi-jointed line.
Notes:	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**.

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.
Purpose:	Draw a series of markers.
Notes:	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.
Purpose:	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.

Purpose: Captures all graphics in the current window to a PostScript file.

Routine PRINT.SEG.R given SEGMENT.ID, USE.DIALOG yielding SUCCESS

Arguments:

SEGMENT.ID	Integer. Segment identifier.
USE.DIALOG	Integer. Nonzero if dialog should be shown.
SUCCESS	Integer. Nonzero if printing was completed.

Purpose: Print the segment identified by **SEGMENT.ID.**

Notes: If **USE.DIALOG** is nonzero the system print dialog is displayed allowing the user to set print options.

Routine PRINT.WINDOW.R given WINDOW.PTR, USE.DIALOG yielding SUCCESS

Arguments:

WINDOW.PTR	Pointer to a window's 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.

Purpose: Prints a window.

Notes: If **USE.DIALOG** is nonzero the system print dialog is displayed allowing the user to set print options.

Routine READ.GLIB.R (FILE.NAME)

Arguments:

FILE.NAME	Text. File name of the graphics library.

Purpose: Read a graphics library file from disk.

Notes: This routine will read a graphics library file created by SIMSCRIPT Studio. 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 at the beginning of execution.

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 = Do not draw any special cursor
- 1 = Draw rubber band line while waiting
- 2 = Draw rubber box while waiting
- 3,4 = Allows a global variable **`DINPUT.V`** to be assigned be repeatedly updated with a new **`LOCATION.A`** value, thus tracking the mouse until a button is clicked.
- 16 = May be used within a SIMSCRIPT process routine (which **`READLOC.R`** will suspend). The mouse position will be sampled from the timing mechanism, allowing it 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.

Purpose: Wait for the user to click somewhere in the canvas with the mouse.

Notes: The graphic cursor is anchored at the given (**`POSX,POSY`**). As the user moves the mouse, the cursor updates automatically. After a click, **`READLOC.R`** scans the viewing transformations in reverse numerical order - from 15 to 0 - until the NDC position can be reverse-transformed. If the mouse is located within a viewport specified by some transformation number, this number is returned. In this way, movement of the mouse between viewports 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	Real. (Currently ignored)
Y	Real. (Currently ignored)
MAXLEN	Integer. (Currently ignored)

Purpose: Read graphic text.

Notes: 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.

Purpose: Provide the names of all loaded graphical objects.

Notes: Returns an array containing names of all graphical objects that have been loaded so far. 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, width, and color. (see **LINEWIDTH.R, LINESTYLE.R, LINECOLOR.R).**
- 1 = Draw a sector of a circle, fill with current fill style and fill color. (see **FILESTYLE.R, FILLCOLOR.R**)

Purpose: Draw an arc or a sector of a circle.

Notes: The first point identifies the center of a circle. The second point should be any point on the circumference and marks the beginning of the arc. An arc is drawn counter-clockwise to the third point.

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.

Purpose: Compute the bounding box of any existing segment.

Notes: 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**.

Purpose: Removes image of a segment and causes a new image to be drawn.

Notes: An assignment to this attribute will delete the previous segment, if one exists. Assigning the value 0 to this attribute will erase the segment.

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 re-displayed at the position and rotation indicated by the **LOCATION.A** and **ORIENTATION.A** attributes.

Global Variable SEGID.V

Mode: Integer. Segment identifier.

Notes: 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 should range between 0 and 255.

Subscript: Pointer to a GRAPHIC entity or to a DYNAMIC GRAPHIC entity.

Notes: The priority of a graphic entitie is supplied through this attribute. Graphic entities 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 SIMSCRIPT. Segments of this priority are not automatically re-displayed by the system.

Routine `SET.ACTIVATION.R (FORM.PTR, ACTIVATE)`

Arguments:

`FORM.PTR`	Pointer to any form or form field.
`ACTIVATE`	Integer: 0 = Deactivate or "gray out" the field. 1 = Activate the field; make it selectable.

Purpose: Set activation state (gray out) of a form or field.

Notes: Sets the activation state of a field on a form. A deactivated field will appear “grayed 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.

Purpose: Sets the cursor status to busy or normal and changes its shape 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 be positioned at the top of the list window.

Purpose: Scrolls the given list box so that the item indexed by **`TOPINDEX`** appears at the top of the list.

Routine `SETVIEW.R (V.XLO, V.XHI, V.YLO, V.YHI)`

Arguments:

V.XLO	Integer, in Normalized Device Coordinates, (where 0 ≤ **V.XLO** < **V.XHI** ≤ 32767).
V.XHI	Integer, in Normalized Device Coordinates.
V.YLO	Integer, in Normalized Device Coordinates.
V.YHI	Integer, in Normalized Device Coordinates.

Purpose: Sets viewport of the current viewing transformation.

Notes: This routine defines a rectangular viewport region on the display surface. The global variable VXFORM.V should be assigned prior to calling this routine. Areas, lines, and points outside this region are clipped. For the purpose of specifying viewports, a separate coordinate system is used where 0 ≤ x ≤ 32767 and 0 ≤ y ≤ 32767.

Routine SET.WINCONTROL.R given WINDOW.PTR, CONTROL.ROUTINE

Arguments:

WINDOW.PTR	Pointer to the window display entity.
CONTROL.ROUTINE	Name of the routine to call when a window event occurs.

Purpose: Specifies the control routine to be used for the given window.

Notes: The given control routine will be invoked 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 SIMSCRIPT window returned by **OPENWINDOW.R.**

Purpose: Associates the current viewing transform (**VXFORM.V**) to the window with the given id.

Notes: 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**).

Purpose: 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.

Notes: Sets the mapping of problem-oriented coordinates, given in real world coordinates. The given arguments define the coordinate system for the rectangular viewport defined using the **SETVIEW.R** routine. If icons are loaded from SIMSCRIPT Studio, the arguments passed to **SETWORLD.R** should agree with the world coordinate system indicated in the "Icon Properties" dialog box of the icon editor. Usually **W.XLO < W.XHI** and **W.YLO < W.YHI**. However, **SETWORLD.R** can be used to invert or mirror-image a transformation by reversing one or both of the above inequalities.

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.

Purpose: 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.
0 = left justified (the default)
1 = centered text
2 = right justified
VERT Integer. Value = 0, 1, 2, 3, or 4.
0 = bottom justified (the default)
1 = centered vertically
2 = top justified
3 = bottom of character cell
4 = top of character cell

Purpose: Set portion of character that is aligned upon the graphic text position.

Notes: 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.

Purpose: Sets the angle of rotation of subsequent lines text.

Notes: An angle of zero represents normal, horizontal text. Line of text is rotated counter-clockwise as the angle increases.

Routine TEXTCOLOR.R (COLOR.INDEX)
Arguments:

COLOR.INDEX	Integer. **COLOR.INDEX** is an integer with values from 0 to 255.

Purpose: Set color of subsequent characters.

Routine TEXTFONT.R (FONT)
Arguments:

FONT Integer. Indicates which font to use.

- 0—Basic font
- 1—Arial
- 2—Roman
- 3—Bold Roman
- 4—Italic
- 5—Script
- 6—Greek
- 7—Gothic

Purpose: Set text font of subsequent characters.

Notes: Calling this routine will specify that a predefined vector font is to be used. These fonts are scaled with the graphics window and can always be rotated, but are defined by SIMSCRIPT and may not match fonts found on your operating system. To show text in one of the fonts found on your system, use the **TEXTSYSFONT.R** routine.

Routine TEXTSIZE.R (SIZE)
Arguments:

SIZE	Integer. The character height in NDC units, with a range of 0 to 32767.

Purpose: Set size of subsequent characters.

Notes: This routine applies to vector fonts only. (see **TEXTFONT.R**)

Routine TEXTSYSFONT.R given FAMILY.NAME, POINT.SIZE, ITALIC, BOLDFACE

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 Integer. Denotes the thickness of the font. Range is 0 to 1000.

ITALIC Integer. Denotes the slant of the font. Range is 0 to 1000.

Notes: Allows programmatic selection of a system (raster) font. These fonts are generally provided by your operating system and availability may vary across platforms. Text drawn using a raster font will remain the same size regardless of how big the graphics window is. If called, the font set using **TEXTFONT.R** is temporarily ignored.

System Global Variable TIMESCALE.V

Mode: Integer.

Purpose: Scales Real time (1/100 second) per simulated time unit.

System Global Variable TIMESYNC.V

Mode: Subprogram variable.

Notes: 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).**

Purpose: Associate a constant velocity with a dynamic graphic entity.

Notes: 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.

Purpose: Set a present velocity given absolute velocity and angle.

Notes: 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.

Purpose: 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.

Purpose: Inquire the present velocity, in the Y direction.

Global Variable VXFORM.V

Mode: Integer. Values range between 0 and 15, inclusive.

Purpose: Indicates which viewing transformation is in effect.

Notes: The default transformation is provided by **VXFORM.V = 0**, a one-for-one mapping of real world coordinates into Normalized Device Coordinates. **VXFORM.V** indicates which transformation is to be defined, redefined, or used. This allows SIMSCRIPT to provide multiple mappings between real-world spaces and areas on the display screen. Such user-defined mappings are specified by first assigning **VXFORM.V** to a unique value, then calling routines like **SETWORLD.R** and **SETVIEW.R**. **VXFORM.V** can be 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.

Purpose: Write a text string to the currently open segment

Notes: 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.

INDEX

V

W

X

Z