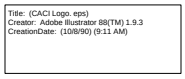
SIMGRAPHICS II®
User's Manual for MODSIM III



Products Company

3333 North Torrey Pines Court, La Jolla, California 92037 • (619) 824.5200 • Fax (619) 457-1184
Watchmoor Park, Riverside Way, Camberley, Surrey GU15 3YL, UK • 1276 671 671 • Fax 1276 670 677
1600 Wilson Blvd., 13th Floor, Arlington, Virginia 22209 • (703) 875-2900 • Fax (703) 875-2904

For product information or technical support contact:

In the US and Pacific Rim:

CACI Products Company
3333 North Torrey Pines Court
La Jolla, California 92037
Phone: (619) 824.5200
Fax: (619) 457-1184

In Europe:

CACI Products Division
Watchmoor Park
Riverside Way
Camberley, Surrey
GU15 3YL, UK
Phone: 1276 671 671
Fax: 1276 670677

Contents

Figures

Preface

This Document

This manual is intended to both teach and serve as a user's manual for SIMGRAPHICS II. SIMGRAPHICS II is a graphical tool kit built on MODSIM III. Using SIMGRAPHICS II you can easily incorporate animation, presentation graphics and graphical user interfaces into your MODSIM III programs. Some familiarity with MODSIM III is assumed.

Free Trial & Training

SIMGRAPHICS II is available exclusively from CACI Products Company. MODSIM III can be sent to your organization for a free trial. We provide everything needed for a complete evaluation on your computer: software, documentation, sample models, and immediate support when you need it.

Training courses in MODSIM III are scheduled on a recurring basis in the following locations:

La Jolla, California
Washington, D.C.
London, United Kingdom

For information on free trials or training, please contact the following:

In the U.S. and Pacific Rim:

CACI Products Company
3333 N. Torrey Pines Ct.
La Jolla, CA 92037
(619)824.5200
Fax (619) 457-1184

In Europe:

CACI Products Division
Watchmoor Park, Riverside Way
Camberley, Surrey GU15 3YL
United Kingdom
1276 671 671
Fax 1276 670 677

Chapter 1: Introduction to SIMGRAPHICS II

The MODSIM III Graphics package allows easy access to animation, presentation graphics, and user-interface toolkits using a graphics editor (SIMDRAW) to simplify construction.

Animation is produced by drawing objects using SIMDRAW and then animating them within a MODSIM III program. Operations such as scaling, rotating, and positioning can be performed. The animated objects or *images* can have subcomponents which move along with the whole object, but can be manipulated individually. For example, a dump truck can have a bed subcomponent which is rotated independently of the base of the truck.

Presentation graphics or *graphs* such as pie charts, level meters, bar graphs, etc. are also created using SIMDRAW, and are then used within the MODSIM III program by asking the graphs to plot values. The visual appearance of the graph is updated automatically.

Access to the user-interface toolkits or *forms* allows input using menu bars, dialog boxes and palettes. The appearance of the forms conforms to the style of the system MODSIM III is running on. On workstations, MOTIF dialog boxes are used; on Microsoft Windows systems, Windows dialog boxes, pull down menus and palettes are produced.

All elements of the MODSIM III graphics library are portable, which means if you take a MODSIM III Graphics program which runs on one system and move it onto a new system it will run, without modifying any code. The forms will change their appearance to conform to the new system they are running on. The images and graphs will have the same appearance as on the previous system.

1.1 SIMGRAPHICS II Environment

The world in which SIMGRAPHICS II lives can be described in terms of a screen and windows. The computer screen can have multiple windows. Each window can contain images, graphs, and forms.

The windows are provided by the system and can be moved and resized. When the window is resized, its contents always maintain the same height to width relationship or aspect ratio.

Graphic images created in the editor can be placed inside the window. The images can be used to provide a background or they can be animated.

Each window can have a menu bar for making menu selections, and multiple dialog boxes for accepting input of various types, such as numerical values, yes/no responses and text. Windows can also contain palettes for changing modes or selecting operations.

1.2 Interface

Since MODSIM III is an Object Oriented Language, the graphical interface is implemented using objects. Images, graphs, and forms are associated with MODSIM III objects. Therefore, using graphics involves creating and manipulating objects. Many basic objects have already been provided. The most important are :

`WindowObj`—Standard system window which can be moved, resized, etc. Acts as a container for all graphical objects

`ImageObj`—Basic object used for static icons and backgrounds.

`DynImageObj`—Basic graphic object used for animation.

`DialogBoxObj`—Receives various types of input from the user. Controls, such as buttons, check boxes, list boxes, radio boxes, value boxes and text boxes can be part of a dialog box.

`MenuBarObj`—Receives simple menu selections.

`PaletteObj`—Receives input from two-state palette buttons.

Since there can be multiple windows on the screen, images, dialog boxes, and menu bars must be added to a specific window. This is done using the **`AddGraphic`** method of the window. For instance, to add an image to a window the method call would be:

```
ASK MyWindow TO AddGraphic(MyImage);
```

To make an object visible on the screen you must ask it to draw:

```
ASK MyWindow TO Draw;
ASK MyImage  TO Draw;
```

To erase an object:

```
ASK MyImage TO Erase.
```

All of the objects created by the editor can be saved in a single file. This single file is called a *Library file* and has an associated object in MODSIM III called a **`GraphicLibObj`**. It is used to re-create the objects built in the editor so they can be used within the MODSIM III program.

To use an image, graph or form created by the editor you must do the following:

1. Create a window:

   `NEW(window)`;

2. Create a **`GraphicLib`** object:

   `NEW(GraphicLib)`;

3. Ask it to read the file containing the objects created in the editor:

```
ASK GraphicLib TO ReadFromFile("graphics.sg2");
```

4. Create instances of objects you want:

```
NEW(TruckImage);
```

5. Ask the instances to customize their appearance by copying an object in the library:

```
ASK TruckImage TO LoadFromLibrary(GraphicLib,
    "truck");
```

6. Add the instances to the window:

```
ASK window TO AddGraphic(TruckImage);
```

7. Draw either the window or object:

```
ASK window TO Draw;
```

1.3 A Simple Program

```
MAIN MODULE Example1;

{ This program gets a "truck" created in the editor and
  displays it for 10 seconds. }

FROM OSMod IMPORT Delay;
FROM Graphic IMPORT GraphicLibObj;
FROM Window  IMPORT WindowObj;
FROM Animate IMPORT DynImageObj;

VAR
   GraphicLib : GraphicLibObj;
   window     : WindowObj;
   truck      : DynImageObj;

BEGIN

   NEW(window);

   NEW(GraphicLib);
   ASK GraphicLib TO ReadFromFile("graphics.sg2");

   NEW(truck);
   ASK truck  TO LoadFromLibrary(GraphicLib, "truck");
```

```
   ASK window TO AddGraphic(truck);
   ASK window TO Draw;
   Delay(10);

END MODULE.
```

Chapter 2: SIMDRAW

2.1 SIMDRAW Overview

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

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

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

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

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

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

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

2.2 Running SIMDRAW

SIMDRAW can be started from within the MODSIM III Workbench, or from the command line. Upon execution a main window containing a palette and toolbar is displayed (figure 2-1). The window will contain a listing in the currently loaded SIMGRAPHICS II library. The palette on the left is used to add new objects to the library.

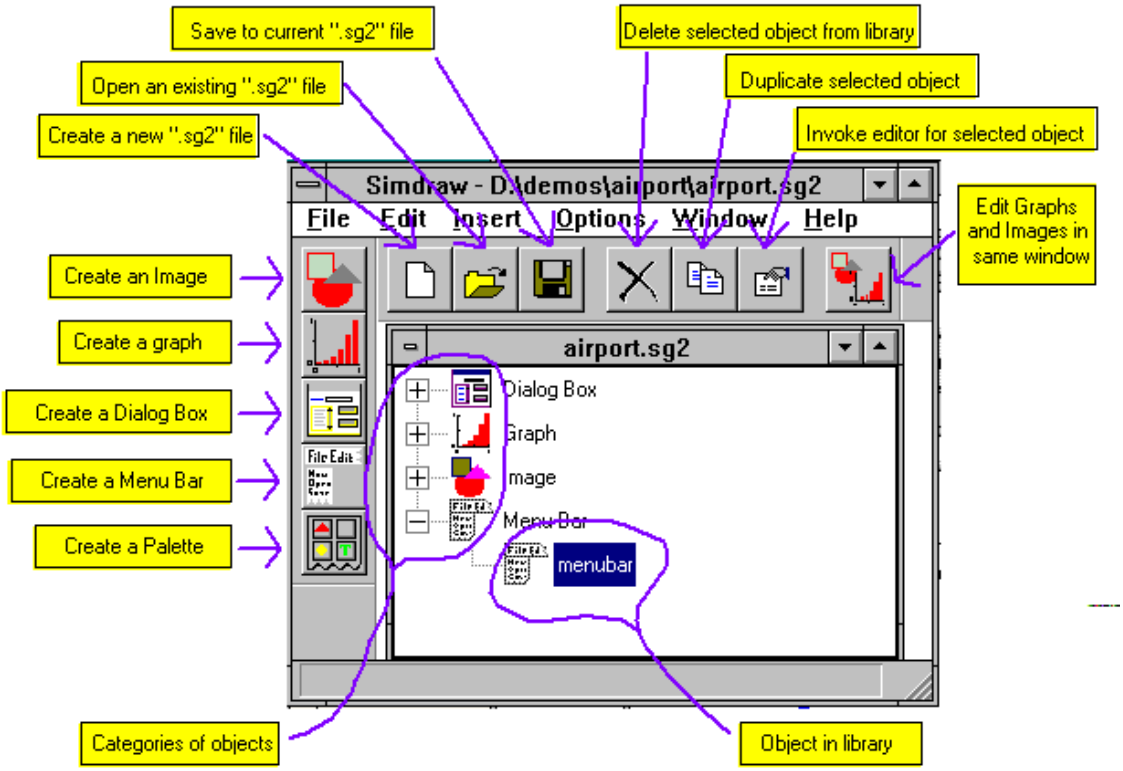


Figure 2-1. Main Window

2.3 Loading and Saving SIMGRAPHICS II files.

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

2.4 Editing an Existing Object

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

2.5 Adding an Object to the Library

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

2.6 Removing an Object from the Library

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

2.7 Making a Duplicate of an Object

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

2.8 Changing the Name of an Object

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

2.9 Adding an Object from Another Library

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

2.10 Editing Images and Graphs in Same Window

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

2.11 Listing of Objects

All objects contained in the current SIMGRAPHICS II library file are shown in a *list window*. Objects shown in this window can be ordered through the **Options** menu option in one of the following three ways:

I. **Natural Order --** Objects are ordered based on time of creation. The objects last added to the library are shown at the bottom of the list.

II. **Alphabetical Order --** Alphabetical order based on name.

III. **Typed Order --** Objects are listed categorically. The categories are: **Image, Graph, Dialog Box, Menu Bar,** and **Palette**. A "heading" is created for each category, with the objects listed alphabetically under the appropriate heading. Click on the (**-**) to the left of the heading to expose the names of the objects.

2.12 Command Line Arguments

SYNOPSIS:

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

The following command line arguments are recognized by SIMDRAW:

`-l file_name`	Specifies the name of the SIMGRAPHICS II graphics file to edit.
`-e, -single`	Specifies "single edit" mode. The specified objects will be edited with no control window containing library information.
`-nodialog`	Eliminates editing of the specified object types
`-nograph`	
`-noimage`	
`-nomenu`	
`-nopalette`	
`-dim xlo ylo xhi ylo`	Specifies the default real world coordinate space used by the **Image Editor**.
`-B path_name`	Specifies the path to bitmap files used by SIMDRAW
`-S path_name`	Specifies the path to system files needed to run SIMDRAW (trailing delimiter '/' or '\' must be included).
`-W path_name`	Specifies path to user Simgraphics II files

2.13 Using the Image Editor

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

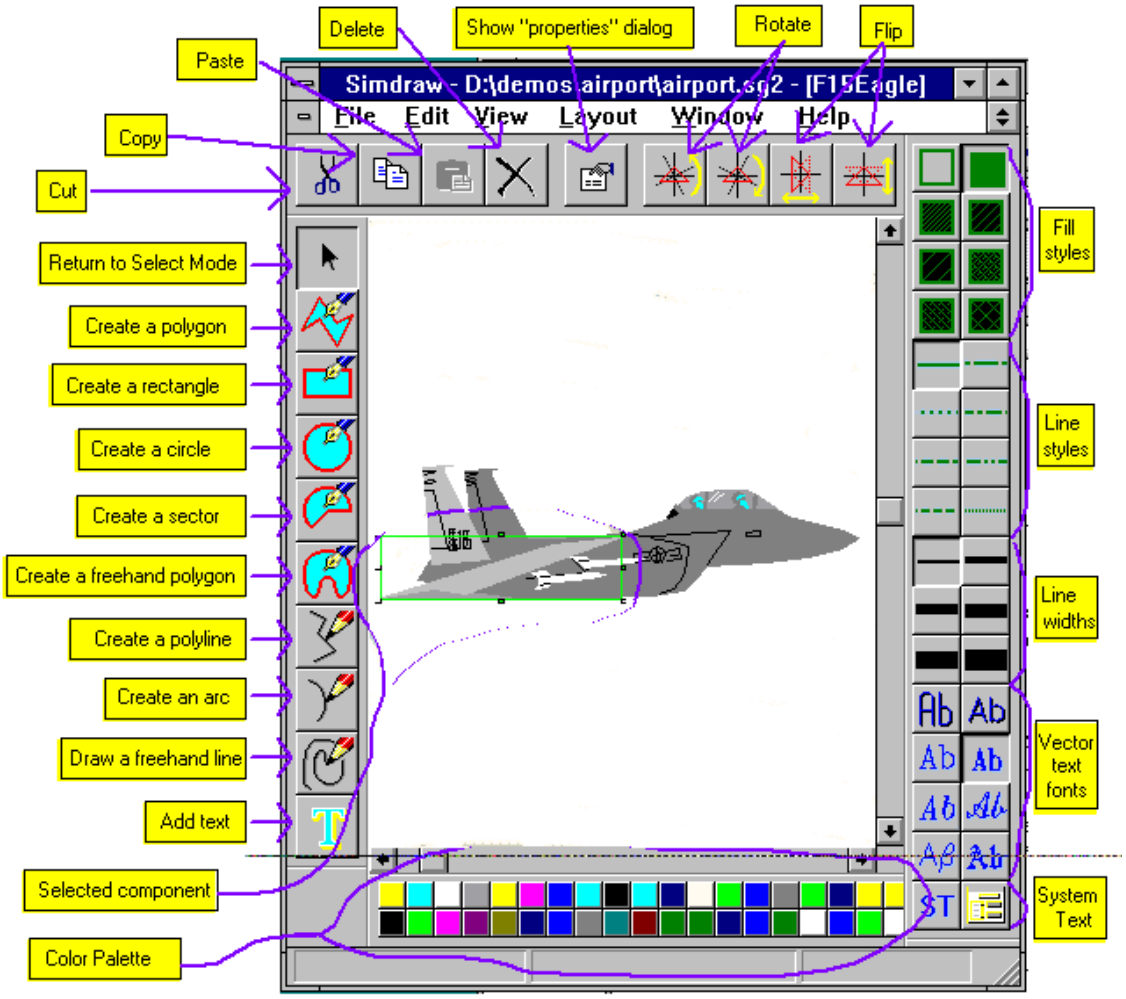


Figure 2-2. Image Editor

2.13.1 Mode, Style, and Color Palettes

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

The **Mode** palette is shown on the left hand side of the **Image Editor** window. Use it to add primitives to your drawing. Refer to paragraph 2.13.4.

2.13.2 Selecting, Moving, and Resizing

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

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

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

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

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

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

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

2.13.4 Creating Primitives

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

Polylines

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

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

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

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

Polygons

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

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

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

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

Circles

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

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

Sectors

A sector is a filled semicircular shape similar to a pie slice. Sectors are composed of a center point, a starting point and an ending point, and are drawn counter clock-wise from the starting point to the ending point. To draw a sector, first press the **Sector** button on the **Mode** palette. Point to where you want the center point of the sector, and drag the mouse. Release the mouse over where you want the starting point of the arc. Drag the mouse to where you want the sector to end and release to return to **Select** mode.

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

Arcs

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

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

Text

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

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

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

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

Bitmaps

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

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

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

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

2.13.5 Creating Images

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

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

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

2.13.6 Editing the Root Image

The editor's window shows all objects contained by the image being edited or the *root* image. To change properties of this image (such as its *name*), de-select all shapes and use the **Edit/Properties** menu option.

To resize the root image, use **Edit/Select All** to select all of its shapes, and then use **Layout/Group** to make a group. The image can be resized by dragging the square resize handles on the green selection box. When the root image is appropriately sized, use **Layout/Ungroup** to eliminate the grouping.

To reset the center point of the root image, deselect all shapes by clicking in the background, and use the **Edit/Recenter** menu option. Select a new center point for the image and then select the **OK** button on the **Recenter** dialog.

2.13.7 Editing Points on a Primitive

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

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

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

2.13.8 Defining Stacking Order or Priority

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

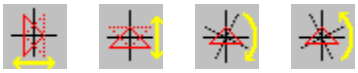
Stacking order is with respect only to other shapes in the same group or image. In other words, the **Bring to Front** menu option will bring the selected shape to the front of all other shapes in that group, but not necessarily to the front of all shapes in the window.

2.13.9 Defining the Center Point of a Shape

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

You can reset the center point of the entire drawing (root image) by de-selecting all shapes and then using the **Edit/Recenter** menu option.

2.13.10 Using the Flip and Rotate Tools



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

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

2.13.11 Align and Distribute

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

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

2.13.12 Using Grid Lines

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

You can change the color of the grid by selecting a color from the **Color** palette and then using the **View/Grid Color** menu option. The granularity of the grid can be adjusted using the **View/Grid Spacing** menu option. Granularity can be **Fine, Medium**, or **Coarse**. The distance between grid lines for **Fine** graduation is 500.0, for **Medium** is 1500.0, and for **Coarse** is 4500.0.

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

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

2.13.13 Changing Views (Panning and Zooming)

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

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

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

2.13.14 Changing Dimension (Coordinate Space Boundaries)

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

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

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

2.13.15 Changing the Layout Size and Color

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

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

2.13.16 Program Access

Any image or primitive added to the root image can be accessed from inside an applic ation by specifying a **Reference** or **Field** name and/or **Id** through the **Properties** dialog box. The image's *library name* should be specified for loading the image into your program.

2.14 Using the Graph Editor

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

2.14.1 Style, and Color Palettes

The **Style** palette on the right hand side of the window contains the set of dash styles, hatch styles, line widths, and text fonts that can be applied to the selected graph components. The **Color** palette on the bottom of the window contains 64 colors that can be applied to a component. When a component is selected, the **Style** and **Color** palettes will be updated to reflect the style and color of that graph part. At this time, **Style** and **Color** palette changes will be applied to the selected part.

2.14.2 Selecting, Moving, and Resizing

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

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

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

2.14.3 Charts (2-D Plots)

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

2.14.3.1 Modifying Chart Attributes

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

- **Library Name** – The name used to load the chart into your application program.
- **Title** – The title shown on the top of the chart.
- **Axes on Edges** – If checked, numbering and tic marks will be forced to appear on the edges of the plot area. For better visual reference, two extra axes will be drawn on both the top and right sides of the plot area.
- **Time Trace Plot** – Setting this item implies that the chart is a time trace plot. Whenever a variable being monitored by the chart is modified, its new value is plotted along the Y-axis and the current *simulation time* is plotted along the X-axis.
- **Show Legend** – Chart will show a legend below the plot area. The fill style and color of each data set is shown preceding its name.
- **Show Border** – A chart can be defined to draw a rectangular background underneath.
- **Data Sets** – A data set can be added using the **Add** button, or removed by selecting its name in the list box and then pressing the **Remove** button. To change the name of a data set, select its current name in the list box and then press the **Edit** button. (see "Attributes of a Data Set")
- **Handling of Multiple Data Sets** – If "stacked", all discrete data sets will be stacked on top of each other. In other words, the value plotted in a data cell is reflected as the *height* of the bar, not its top. Therefore, *stacking* means that the bottom of a cell in data set *n* is equal to the top of the same cell in data set *n-1*. I.e. higher numbered data sets are stacked onto the lower numbered ones.

2.14.3.2 Modifying the X-Axis

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

- **Title** – Label for X-axis displayed below numbering.

- **Rescaleable** – Specifies whether the x-axis will be re-numbered (scaled) when one of the data points extends beyond its limit. In this case, the **Compress Data** item determines whether a scrolling window is used, and whether old data is discarded, or the range of the graph is to be expanded showing all data. Note that re-scaling may modify the tic mark, numbering, and grid line intervals to maintain a similar visual representation of the chart. If this item is not checked, data points falling beyond the limits of the X-axis will be discarded.
- **Show Grid Lines** – If this item is on, *grid lines* will be shown crossing the X-axis.
- **Tics Centered, Tics Inside, Tics Outside** – Defines the tic mark alignment with respect to the X-axis line. Tics marks can be attached to the X-axis from their center, left or right sides.
- **Compress Data** – When this item is set, re-scaling the X-axis will increase the coordinate area of the chart enough to encompass the offending data point. As a result, existing data will shrink in size. Clearing this item will have data *scrolled* along the X-axis during axis rescale. In this case, data scrolled out of view will be discarded.
- **Minimum, Maximum** – Defines the initial X-axis data range of the chart.
- **Tic Interval (Major & Minor)** – Defines the distance along the X-axis between consecutive tic marks. If an interval is zero, tic marks will not be displayed.
- **Numbering Interval** – Defines the distance along the X-axis between consecutive number labels on the axis.
- **Grid line Interval** – Defines the distance along the X-axis between consecutive grid lines.
- **Y Intersection Point** – Defines the point (in x-axis coordinates) along the X-axis where the Y-axis intercepts.
- **Y2 Intersection Point** – Defines the point (in x-axis coordinates) along the X-axis where the second Y-axisintercepts.
- **Data Scaling Factor** – Defines the factor multiplied to the X component of all data plotted to the chart at runtime.

2.14.3.3 Modifying the Y-Axis

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

- **Title** – Label for Y-axis displayed to the left of its numbering.
- **Rescaleable** –Specifies whether the Y-axis will be re-numbered (scaled) when one of the data points extends beyond its range. Note that re-scaling may modify the tic mark, numbering, and grid line intervals to maintain a similar visual representation of the chart. If this item is not checked, data points falling beyond the limits of the Y-axis will be clipped.

- **Show Grid Lines** – If this item is on, *grid lines* will be shown crossing the Y-axis.
- **Tics Centered, Tics Inside, Tics Outside** – Defines the tic mark alignment with respect to the Y-axis line. Tics marks can be attached to the Y-axis from their center, left or right sides.
- **Minimum, Maximum** – Defines the initial Y-axis data range of the chart.
- **Tic Interval (Major & Minor)** – Defines the distance along the Y-axis between consecutive tic marks. If an interval is zero, tic marks will not be displayed.
- **Numbering Interval** – Defines the distance along the Y-axis between consecutive number labels on the axis.
- **Grid Line Interval** – Defines the distance along the Y-axis between consecutive grid lines.
- **X Intersection Point** – Defines the point (in y-axis coordinates) along the Y-axis where the X-axisintercepts.
- **Data Scaling Factor** – Defines the factor multiplied to the Y component of all data plotted to the chart at runtime.

2.14.3.4 Modifying the Second Y-Axis

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

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

- **Data Scaling Factor** – Defines the factor multiplied to the Y component of all data plotted to the chart at runtime.

2.14.3.5 Attributes of a Data Set

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

- **Representation** – Defines how the overall data set is structured. You can choose one of the following data set types:
  1. **Bar Graph** – Contains a fixed number of cells. Each new data point changes the nearest cell's plot. Neighboring cells are NOT connected. The first cell begins at (X_Minimum - Cell_Width / 2) units. The individual bar is centered over the cell, and there is a small gap between bars.
  2. **Histogram** – Also contains a fixed number of cells. Each new data point changes the nearest cell's bar. There is no connection between neighboring cells. The bar is set at the left edge of the cell, and there is no gap between bars. The first data cell begins at the X-axis minimum
  3. **Discrete Surface** – Neighboring cells are connected to form a surface, however there are still a fixed number of cells. Each new data point changes the nearest "peak or valley" on the surface. The first cell begins at (X_Minimum - Cell_Width / 2) units.
  4. **Continuous Surface** – Variable number of cells, i.e. a new cell is added to the graph each time a data point is plotted at the given (x,y) location. Neighboring cells are connected.

- **Plot Type** – A data set can be shown using a filled region or a simple surface line with or without markers:
  1. **Fill** – Plot a data cell using a filled polygon. The fill style can be reset using the **Style** palette .
  2. **Line** – Plot data cell using a polyline. Use the **Style** palette to reset the line width or dash style.
  3. **Marker** – Use a small marker to represent the data point. The specific marker used for the data point is determined from the **Edit/Mark Style** menu option. Markers are only valid for the "continuous surface" representation.

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

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

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

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

2.14.3.6 Creating a Time Trace Plot

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

2.14.4 Pie Charts

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

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

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

2.14.5 Clocks

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

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

2.14.6 Dials

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

- **Library Name** – The name of the object within the current graphics library.
- **Title** – Text of title displayed on bottom.
- **Minimum, Maximum** – Defines the range of values shown by the dial.
- **Interval** – Distance between tic marks around the face.

- **Num Interval** – Distance between numbers around the face
- **Min Theta** – Angle in degrees where the minimum value is placed around the dial circumference.
- **Max Theta** – Angle in degrees where the maximum value is placed around the dial circumference.
- **Scale Factor** – Factor multiplied by value before being displayed in the dial.
- **Show Border** – A square background can be shown under the dial face and title.

2.14.7 Level Meters

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

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

2.14.8 Digital Displays

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

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

- **Scale Factor** – Factor multiplied by value before being displayed in the meter.

2.14.9 Text Meters

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

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

2.15 Using the Dialog Editor

The **Dialog Editor** (figure 2-3) provides a fast and easy to use drag and drop facilities for creating and editing dialog boxes. A dialog box is a container for controls which accept various types of input. A dialog box can contain buttons, single and multi-line text boxes, combo boxes, value boxes, list boxes, radio boxes, check boxes, text labels, and tables. Tabbed dialog boxes can also be created. Items contained by a dialog box or a dialog box tab are called *controls*.

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

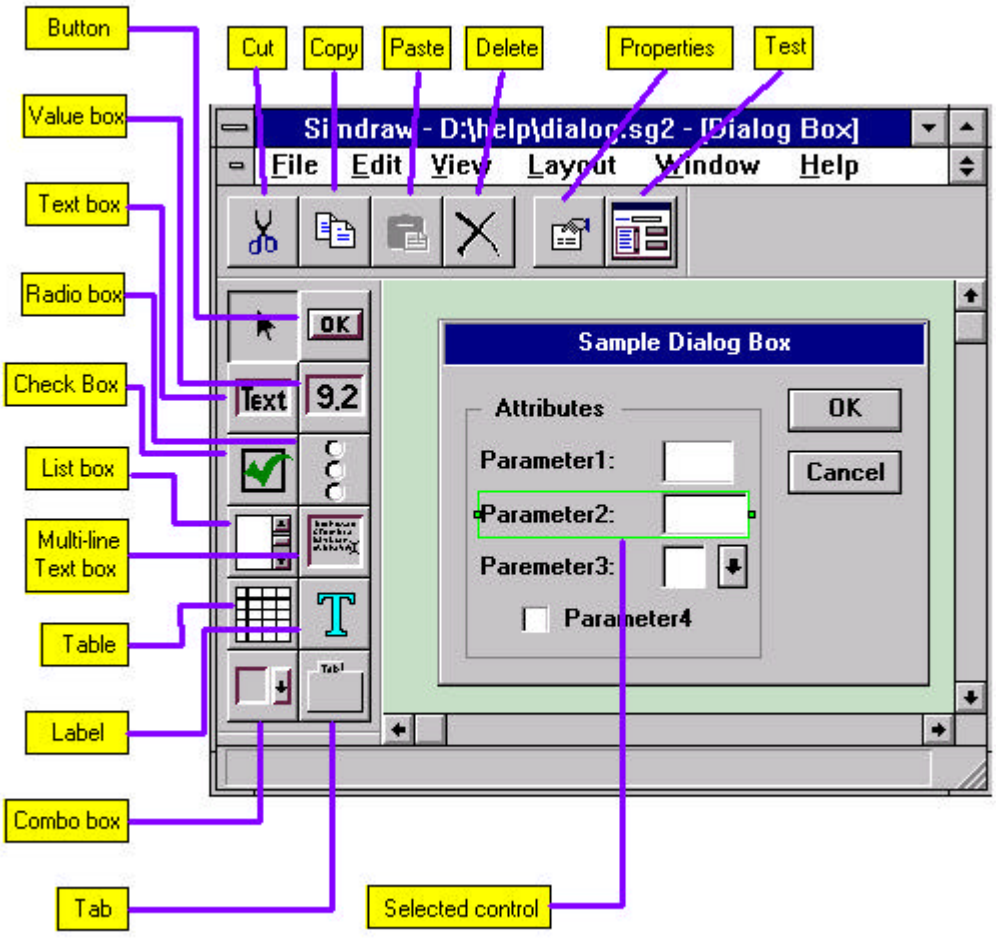


Figure 2-3. Dialog Editor

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

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

2.15.1 Selecting, Moving, and Resizing

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

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

2.15.2 Dialog Box Coordinate System

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

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

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

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

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

2.15.4 Controls

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

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

Buttons

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

- **Label** – This is the text shown on the face of the button

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

Text Boxes

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

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

Value Boxes

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

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

Check Boxes

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

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

Radio Boxes

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

- To add a button, enter its *label*, *reference name*, and *id* 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 buttons, select its label in the list box, modify its label, reference name, or id, and then press the **Update** button.

List Boxes

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

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

Multi-line Text Box

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

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

Labels & Group Boxes

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

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

Combo Boxes

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

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

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

Tables

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

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

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

- **Viewed Width** – The total width in font units of space occupied by the entire table (including row headers, and scroll bar).
- **Viewed Height** – The total height in font units of space occupied by the entire table (including column headers and scroll bar).
- **Number Columns** – Number of columns of cells (not including headers).
- **Number Rows** – Number of rows of cells (not including headers).
- **Column Headers** – If checked, the table will contain a separate row of column headers at the top of the cells.
- **Row Headers** – If checked, the table will contain a separate column of row headers on the left of the cells.
- **Automatic Grow** – If checked, the table will automatically add a row, if the you attempt to move past the last row with the down-arrow key.

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

- **Column (1,2,...) Width** – The number of characters shown in the cells of a particular column. Select the cell in the column corresponding to the one you want to change, and type in a new width.

- **Column (1,2,...) Alignment** – Text in a table cell can be justified to the left or right, or can be centered. Within the **Column Detail** table (**l**=Left justified, **c**=Centered, and **r**=Right justified).

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

Dialog Box

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

- **Library Name** – The name used to access the dialog box from inside the application.
- **Title** – The text shown on the header bar of the dialog.
- **Modal Interaction** – Defines whether the dialog is "modal" or "modeless". When a modal dialog box is displayed, the user cannot interact with any other component of the application but the contents of that dialog box. Modeless dialogs can be interacted with asynchronously.
- **Position with Respect to Screen** – Specifies which corner of the screen the dialog box will be offset from. For example, if **Bottom Left** positioning is selected, the **X Offset** and **Y Offset** fields define the distance from the bottom left hand corner of the screen to the bottom left 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 Ordering of Members** – If you wish to use the **Tab** key to transfer input focus from one control to the next while interacting with the dialog box, the order in which this traversal takes place can be established ahead of time. A list box shows the labels of all controls in the dialog that can have input focus. The order of items in this list is the order in which input focus will proceed when the **Tab** key is pressed. Use the up- and down-arrow keys to shift the tab ordering of controls.

Tabbed Dialogs

The **Dialog** editor can be used to create **Tabbed Dialogs** or to convert existing dialogs to be tabbed. Using a **Tabbed Dialog** you can attach sets of controls to overlapping **Tab Fields**. Only the top **Tab Field** can be seen; all other tab fields and attached controls

remain hidden underneath. The only visible portion of a **Tab Field** is a small rectangular area containing its name, or a *tab* . Clicking on the tab will bring the **Tab Field** to the top of the tab area and show all controls attached to it.

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

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

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

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

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

2.15.4.1 Converting Conventional Dialog Boxes to be Tabbed

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

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

2.15.4.2 Align and Distribute

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

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

2.15.4.3 Using Grid Lines

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

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

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

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

2.15.4.4 Changing Views (Panning and Zooming)

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

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

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

2.15.4.5 Changing the Layout Size, Color and Font

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

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

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

2.16 Using the Menu Bar Editor

A menu bar contains *menus* which can contain either menu items, or other menus. The **Menu Bar Editor** (figure 2-4) allows you to construct a menu bar by interactively dragging and dropping icons representing menus and menuitems onto a menu bar icon.

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

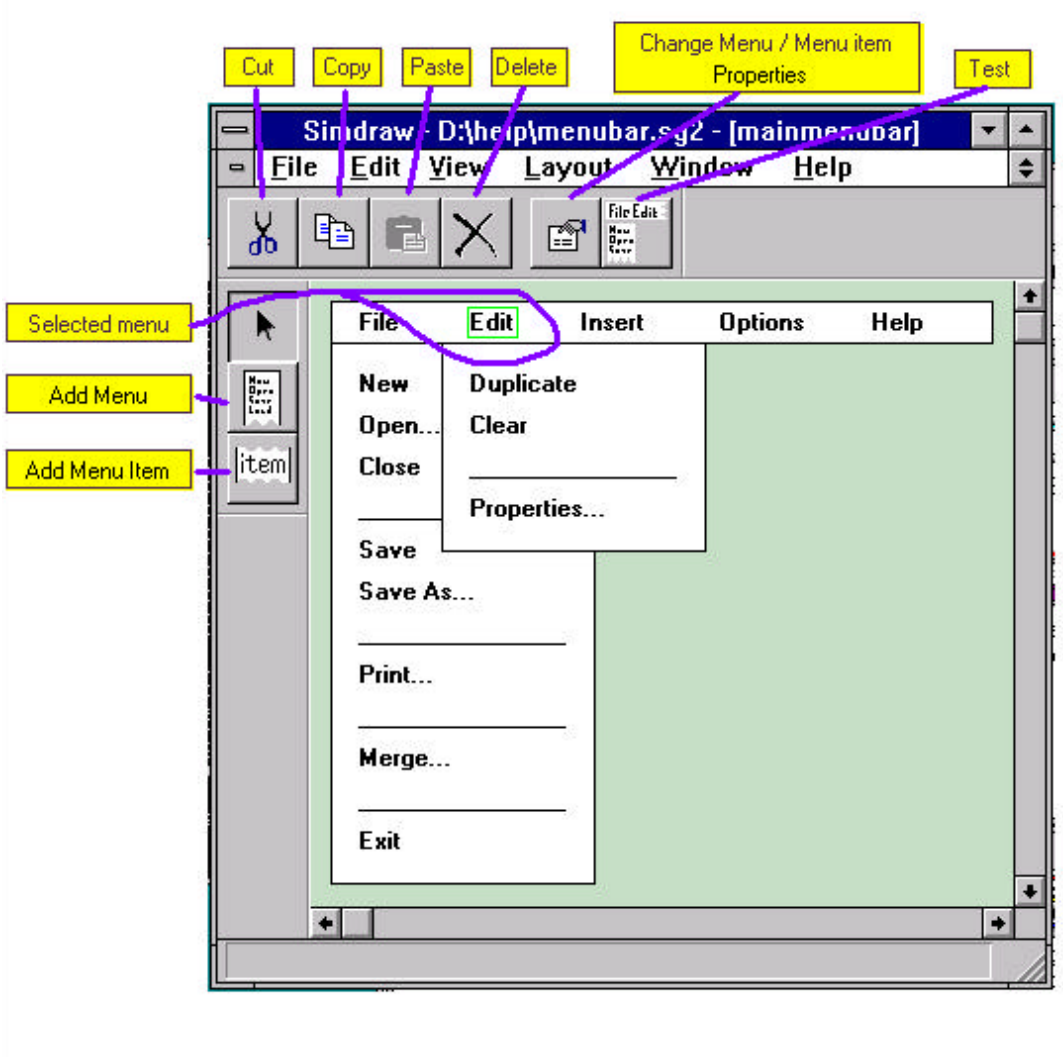


Figure 2-4. Menu Bar Editor

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

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

2.16.1 Selecting and Moving (Transferring)

A menu or menu item can be selected by clicking the mouse button over its label. S elected menus are marked by a bordering green or cyan box. Selecting the label of a menu will drop down its pane, showing all the items it contains. Multiple items can be selected by

holding down the **Shift** key and then clicking on several items. To add a menu or item to another menu, drop it onto the menu's open pane.

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

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



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

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

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

2.16.3 Editing the Menu Bar

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

2.16.4 Editing a Menu

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

- **Reference (Field) name, Id** – Any menu added to the menu bar or another menu can be accessed from inside an application by specifying a **Reference** or **Field** name and/or **Id**. For SIMSCRIPT II.5 users, the field name is passed to the callback routine whenever a menu item is clicked on.

- **Label** – The name identifying the menu which appears within the container menu bar or menu.
- **Mnemonic** – A letter in the menu's label that can be typed from the keyboard (while holding down the **Alt** key) to bring down the menu pane. The mnemonic character will appear underscored in your application.

2.16.5 Editing a Menu Item

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

- **Reference (Field) name, Id** – Any menu item can be accessed from inside an application by specifying its **Reference** or **Field** name and/or **Id**. For SIMSCRIPT II.5 users, 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 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 keys such as [a-z], [0-9], and other punctuation and symbol keys to activate the menu item, 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.

2.17 Using the Palette Editor

The **Palette Editor** (figure 2-5) provides a fast and easy to use drag and drop facility for creating and editing palettes, toolbars etc. A palette is usually attached to the side of your window (but is sometimes a separate window) and contains an array of buttons. The face of each button can contain a bitmap icon or show a color. Separator objects can be added to the palettes to produce space between groups of buttons.

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

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

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

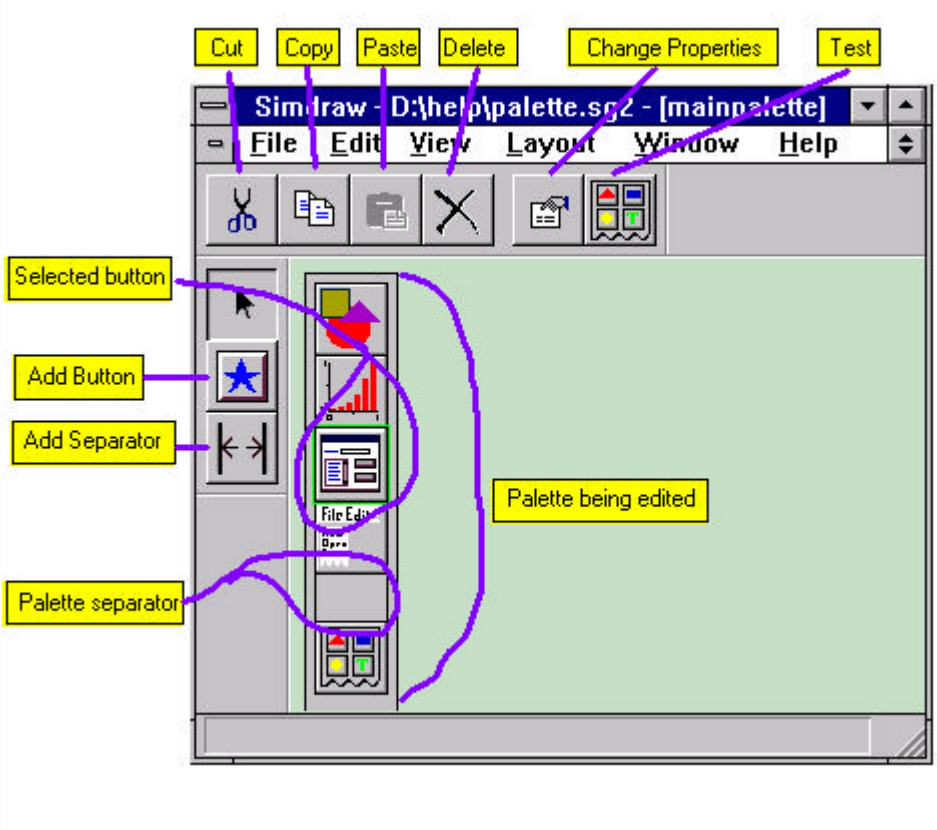


Figure 2-5. Palette Editor

2.17.1 Selecting and Moving (Rearrangement) of Buttons

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

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

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

2.17.2 Using the Clipboard (Cut, Copy and Paste)



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

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

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

2.17.3 Editing the Palette

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

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

2.17.4 Editing a Palette Button

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

- **Reference (Field) Name, Id** – Any button added to the palette can be accessed from inside an application by specifying a **Reference** or **Field** name and/or **Id.** For SIMSCRIPT II.5 users, the field name is passed to the callback routine whenever the button is clicked on.
- **Icon Name** – The name of the bitmap resource or file (without extension) icon displayed on the front of the palette button. Pressing the small browse "**..**" button next to this text box will allow you to browse the file system to select a bitmap file name. Remember that the bitmap file MUST be in the same directory as your library (`.sg2`) file.
- **Status Message** – Text displayed in pane 0 of the parent window's status bar (if present) whenever the pointer passes over this button.
- **Tool Tip** – Identifies the tool tip pop up message shown at the pointer's current location when it passes over this button
- **Momentary/Draggable/Toggle** – Determines the variety of input interaction. One of three button types can be selected:

  1. **Momentary** – Button will automatically pop back up after it is pressed.
  2. **Toggle** – Two state button. The state (up or down) alternates with each activation.
  3. **Draggable** – Like **Toggle** but allows you to hold the mouse button down and drag an outline of the palette button bitmap onto the window.

- **Icon Button/Color Button** – If the **Icon Button** item is activated, the face of the palette button will show the bitmap defined by the **Icon Name** field above. For **Color Buttons** the button will be colored using the RGB parameters defined below.
- **Button Face Color (Red,Green,Blue)** – You can set the color of the **Color Button**s through these value boxes. Color is defined by the percentage of Red, Green, and Blue (range [0-100]).

2.17.5 Editing Palette Separators

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

Chapter 3: Graphic Objects and Methods

The MODSIM III graphic library provides animation, presentation graphs, and user input forms. These three broad object categories have overlapping requirements. For instance, you need to be able to draw and erase all of them.

The graphic library was written in MODSIM III to take advantage of MODSIM III's object oriented facilities. The library is built around a few objects which encapsulate the attributes common to many different objects, and pass on these attributes to less fundamental objects by inheritance.

A number of objects have been constructed to pass on their fundamental attributes, and are not intended to be used directly. Such objects are termed *virtual objects*. One way to recognize a virtual object is that they normally end in **`VObj`**. Virtual objects pass on their attributes to more specific objects which you can use more easily.

All MODSIM III graphical objects that can have a physical appearance are derived from the **`GraphicVObj`** object. This object provides the capabilities of drawing, positioning, and selection. Another property of graphical objects is the ability to contain sets of other objects. In other words, all graphical objects can have other objects added to or removed from them.

With an understanding of this basic object in mind we can examine the capabilities of more useful objects that are derived from **`GraphicVObj`**. A **`WindowObj`** represents a window in the computer screen. Since it is derived from **`GraphicVObj`**, it can be drawn, positioned, selected, or have objects added to or removed from it.

An **`ImageObj`** is an object appearing on the canvas of a window. It has the ability to be scaled, rotated, and to have color. Since it is derived from **`GraphicVObj`**, it can be drawn, positioned, and selected as well.

Graphical objects which use the vendor based tool kits to receive input are derived from **`ControlVObj`**. Controls which act as containers are derived from a **`FormVObj`**. The two types of forms are **Dialog boxes** and **Menu bars.**

3.1 Properties of Graphical Objects

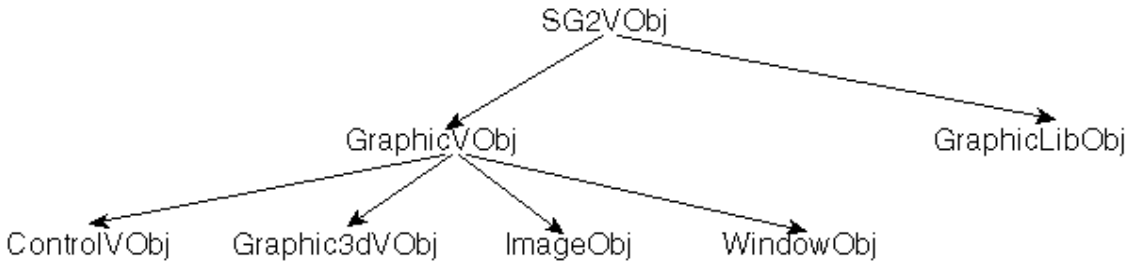


Figure 3-1. Inheritance Tree for Graphical Objects

A property of graphical objects is that they contain a set of other objects. The methods used to add and remove objects from these sets are similar to those of a MODSIM III **QueueObj**. The names these methods are identical to **QueueObj** names but have had the word **Graphic** suffixed to them.

There are three additional methods of a **GraphicVObj** that are not present in a **QueueObj**. These are **AddChild**, **RemoveChild** and **Child**. **AddChild** takes a graphical object, an **Id** and a **ReferenceName** as its arguments. The graphical object's **ReferenceName** and **Id** fields are set to these arguments. These fields are used to later retrieve an object from the list using the **Child** method. This method also takes an **Id** and **ReferenceName** as arguments and returns the graphical object in the list that matches both the **ReferenceName** and **Id**. Note that the reference name and id of a graphical object can be set in the graphics editor. The **Child()** method can be used to get a handle to this object.

The **Descendant()** method is identical to **Child()** except that if no matching child is found, the objects in the CHILDREN's sets will be recursively searched.

For example, suppose you wanted to add the **ImageObj** "truck" to a window, get a handle to its child with the reference name **load**, and then remove the first graphical object in this load:

```
...
ASK window TO AddGraphic(truck);
load := ASK truck Child("load", 0);
firstInLoad := ASK load TO RemoveGraphic();
...
```

Another property of graphical objects is that they can be loaded and saved from graphical object libraries. A **GraphicLibObj** keeps libraries of graphical objects. The **GraphicVObj** methods **LoadFromLibrary()** and **SaveToLibrary()** load and save graphical descriptions to a library. A name (not the reference name) is given to the objects for reference within the library. This is the same name given to the object when it is saved in the editor. For example, this code loads in the graphical description of a "truck" from a library:

```
...
VAR
  truck : DynImageObj;
  library : GraphicLibObj;
...
ASK truck TO LoadFromLibrary(library, "Truck");
...
```

A **GraphicLibObj** is also used to retrieve objects created within SIMDRAW. Within SIMDRAW, you create a library of objects. SIMDRAW saves this library to a file whose name you specify. At runtime you create an instance of a **GraphicLibObj** and ask it to **ReadFromFile** to obtain a copy of this library. Objects can be loaded from the l ibrary with the **LoadFromLibrary ASK METHOD**. As an example, suppose you have created a library of objects within SIMDRAW and named them **"MERCURY"**, **"VENUS"**,

and **"EARTH"**. To load the graphic descriptions of these objects the following block of code could be used:

```
{ Assume file "SolarSys.sg2" was created by
  SIMDRAW and contains the objects named
  "MERCURY", "VENUS", and "EARTH" }

VAR library : GraphicLibObj;
VAR mercury, venus, earth: ImageObj;
...
NEW(library);
NEW(mercury);
NEW(venus);
NEW(earth);
ASK library TO ReadFromFile("SolarSys.sg2");
ASK mercury TO LoadFromLibrary(library,
"MERCURY");
ASK venus   TO LoadFromLibrary(library,
   "VENUS");
ASK earth   TO LoadFromLibrary(library,
   "EARTH");
...
```

3.2 Behaviors of Graphical Objects

When an operation is performed on a graphical object that begins with **Set**, the effect of that operation is not seen visually until the object is asked to **Draw**. Operations not beginning with **Set** have immediate effect. For example, if you wanted to change the position of a graphical object and actually see this position change, the following code could be used:

```
...
ASK graphic TO SetTranslation(x,y);
  { sets the position }
ASK graphic TO Draw();
  { updates object display }
...
```

The **DisplayAt** method is equivalent to the code above. Since **DisplayAt** does not begin with**Set**, the operation takes immediate effect:

```
...
ASK graphic TO DisplayAt(x,y);
...
```

Whenever a graphical object is asked to **Draw,** all of its children will be asked to draw as well. Any objects that were previously erased will now be shown in the window. To maintain visibility status, the **Update** method should be used. This method will redraw only those objects that are currently visible.

Another behavior common to all graphical objects is that they have an automatic asynchronous selection capability. Whenever a leaf graphical object within the tree, is clicked on with the mouse, its **BeSelected** method will be invoked automatically. This method should be overridden to obtain notification of selection. If the **INHERITED BeSelected;** statement is included in this code, then **BeSelected** will be called for the object's parent. If you put the **INHERITED BeSelected** statement at the top of the selection routine for an object, then notification of selection will proceed in a top down fashion. (The selection code for the object's parent will be executed first). Putting the statement at the bottom of the routine will inform in a bottom up fashion. For exa mple, the code:

```
MyDynImageObj = OBJECT(DynImageObj)
 OVERRIDE
  ASK METHOD BeSelected;
END OBJECT;
```

could be used to define an object to be notified of selection. The implementation code for such an object could look like this:

```
ASK METHOD BeSelected;   { top down selection }
BEGIN
  INHERITED BeSelected;  { inform parent of
                           selection }
  OUTPUT("I was clicked on!");
END METHOD;
```

or like this:

```
ASK METHOD BeSelected;   { bottom up selection }
BEGIN
  OUTPUT("I was clicked on!");
  INHERITED BeSelected;  { inform parent of
                           selection }
END METHOD;
```

Chapter 4: Windows

One of the most useful objects within the MODSIM III graphics library is a **WindowObj**. A **WindowObj** represents a window on the screen that contains graphics. All graphic objects including images, forms and graphs must be added to a window before they can be displayed. A**WindowObj** has the following properties:

- *Appearance*—A **WindowObj** can be made visible and invisible using the **Draw()** and **Erase()** ask methods. You can also change its size and position on the screen.
- *Color*—You can change the background color of a **WindowObj**.
- *Contents*—A **WindowObj** can contain images, dialog boxes, a menu bar and palettes.
- *Mouse Monitoring*—A **WindowObj** can receive mouse clicks and monitor mouse movement within its largest centered square.
- *Cursor*—A **WindowObj** can have a cursor that tracks the mouse.
- *Scroll Bars*—A Window can have vertical and horizontal scroll bars. The size and position of the "thumb" on the scroll bar can be set programatically.
- *Status Bar*—A multi-paned "status bar" can be contained by a window. Text in the status bar can be programmatically changed.

4.1 Size and Positioning

The default size and position of a **WindowObj** is the largest possible centered window. The **SetTranslation** method is used to position a **WindowObj** at a specific location on the screen. The **Translation** specifies the position of its lower left hand corner in screen coordinates. The screen coordinate system has (0.0, 0.0) at the lower left hand corner of the screen, and (100.0, 100.0) at the upper right corner.

The **WindowObj ASK METHOD SetSize** sets the size of the window. Size is specified in width and height, and is given with respect to the same coordinate system as described above. For example, to set the size of a window to 50% of the screen height and width, and display the window in the center of the screen, the following code could be used:

```
...
VAR window : WindowObj;
...
NEW(window);
ASK window TO SetSize(50.0, 50.0);
ASK window TO SetTranslation(25.0, 25.0);
ASK window TO Draw();
...
```

4.2 Background Color

The background color of a window can also be set. This color can be set to one of the predefined colors, or be described in terms of its RGB components. The **`SetColor`** and **`SetRGBColor`** methods accomplish this. For example to set the background color of a window to **`Blue`**:

```
...
ASK window TO SetColor(Blue);
ASK window TO Draw();
...
```

4.3 Adding Graphical Objects to a Window

Since a **`WindowObj`** inherits grouping capabilities from **`GraphicVObj`**, it contains a list of graphical objects. There are a number of graphical objects that can be added to a window's list. These include images, menu bars, and dialog boxes. Any number of i mages and dialog boxes can be added to a window, but it can contain only one menu bar. A window can have up to four docked palettes.

For example, if you had a menu bar, dialog box, and an image and wanted them to appear within a window:

```
...
VAR window : WindowObj;
VAR dialogBox : DialogBoxObj;
VAR menuBar : MenuBarObj;
VAR rootImage : ImageObj;
...
ASK window TO AddGraphic(dialogBox);
ASK window TO AddGraphic(menuBar);
ASK window TO AddGraphic(rootImage);
ASK window TO Draw;
...
```

Note: Whenever a window is disposed of, drawn, or erased, all of its contents are also disposed of, drawn or erased.

4.4 Coordinate Systems for Windows

A coordinate system can be set up for a window which its image children will obey. Coordinate systems physically apply to the largest centered square in the window, or to the largest square defined by the width of the window if **`XMajorMap`** is used, or the largest square defined by the height if **`YMajorMap`** is used. (Graphics cannot be drawn beyond the extent of the largest centered square unless the mapping mode for the window is set to **`XMajorMap`** or **`YMajorMap`**). Refer to paragraph 4.5. An image that has no ancestor image containing a coordinate system will be positioned with respect to its window's world. The **`ShowWorld`** method sets the world coordinate system of a window. When this method is called, all images contained in the window will be redrawn with respect to the new coordinate system. Images that are to be added to a window should be saved from SIMDRAW using that window's dimensionality. (Use the **Layout/Dimension** option.) If the **`ShowWorld`** method is never called, then the default world is used. The boundaries of this world are found in the **`GTypes`** module. In this world, **`(WorldXlo, WorldYlo)`** is the lower left-hand corner of the largest centered square, and **`(WorldXhi, WorldYhi)`** is the upper-right corner. The aspect ratio of the screen may be distorted depending on how the graphical objects map to their new distorted world.

One useful method for changing how the contents of a window are viewed is the **`ZoomIn`** method. When given a box in real world (window) coordinates, this method will zoom the contents of that box to become the entire window. The aspect ratio of the scene will be distorted if this box is not square. The **`ZoomIn`** method will automatically redraw the contents of the window. Suppose you wanted to set the world coordinate system of a window to (-100,-100), (100,100) and zoom in to the (0,0), (10,10) square:

```
...
ASK window TO AddGraphic(truck);
ASK window TO ShowWorld(-100.0, -100.0, 100.0,
                                  100.0);
ASK window TO ZoomIn(0.0, 0.0, 10.0, 10.0);
...
```

4.5 Creating Non-square Windows

On occasion you may want to display images on a rectangular, non-square window . In order to be able to do this and write to the entire contents of the window (not-just its largest centered square), you must use the **`SetMappingMode`** method of **`WindowObj`**. This method allows you to specify a major axis of the window that will contain the full width or height of the window's world coordinate space (e.g. 0 .. 32767) while the other axis shows only a portion. The individual modes are described as follows:

`CenteredSquareMap`
(default) The window's contents are viewed in the largest centered square of the window canvas. Graphics cannot be seen outside of this square.

`XMajorMap`
Regardless of window size, the entire X-axis extent of the window's coordinate system can be viewed in exactly the space provided by the window. If the window is wider than it is tall, some the window's contents may be clipped along the top. For windows that are taller than wide, extra non-writable space will be seen at the top. Using this mode, window contents are never clipped along the side, nor is there ever any non-writable space along the side.

`YMajorMap`
Regardless of window size, the entire Y-axis extent of the window's coordinate system can be viewed in exactly the space provided by the window. If the window is taller than it is wide, some the window's contents may be clipped along the right. For windows that are wider than tall, extra non-writable space will be seen at the right. Using this mode, window contents are never clipped along the top, nor is there ever any non-writable space along the top.

Note: The aspect ratio of images seen in a window is never distorted under any mapping mode.

EXAMPLE:

Suppose we wanted to see an image move along the bottom of the entire canvas of a short but wide window:

```
ASK window TO SetTranslation(0.0 40.0);
ASK window TO SetSize(100.0, 20.0);
ASK window TO SetMappingMode(XMajorMap);

ASK dynimage TO SetSpeed(1000.0);
ASK dynimage TO SetTranslation(0.0, 0.0);
```

```
ASK dynimage TO MoveTo(32767.0, 0.0);
```

Using modes **XMajorMap** and **YMajorMap** it is possible that not all of the windows contents can be seen. You can discover the portion the window's coordinate system that is visible using the **ViewableArea** method.

EXAMPLE:

```
VAR xlo, ylo, xhi, yhi : REAL;
...
ASK window TO SetMappingMode(XMajorMap);
ASK window TO SetSize(80.0, 20.0);
ASK window ViewableArea(xlo, ylo, xhi, yhi);
```

You may also want to **SET** the viewable portion of the window; thereby letting you decide what part of the window's coordinate system is visible. This can be done using the **SetAspectRatio** method. The aspect ratio of a window is its width/height. If the mapping mode is **XMajorMap,** the height of the window is modified to reflect the aspect ratio. Otherwise, window width is changed.

EXAMPLE:

Suppose you wanted the viewable portion of the windows contents to be **xlo = 0.0**, **ylo = 0.0**, **xhi = 32767.0**, **yhi = 22000.0:**

```
ASK window TO SetSize(80.0, 80.0);
ASK window TO SetMappingMode(XMajorMap);
ASK window TO SetAspectRatio(32767.0 / 22000.0);
   { window height is changed }
ASK window TO Draw;
```

4.6 Mouse Monitoring

Another property of a window is that it can monitor the mouse. The window (optionally) can be informed of both mouse movement and mouse button clicking. The default is to monitor both. Monitoring of mouse movement is turned on and off by the method. Monitoring of mouse button clicks is turned on and off by the **SetClickMonitoring** method. For example, to tell a window to monitor mouse movement only, the following code is used:

```
VAR window : WindowObj;
...
ASK window TO SetClickMonitoring(FALSE);
ASK window TO SetMoveMonitoring(TRUE);
```

```
...
```

If **ClickMonitoring** is set, the window's **MouseClick** method will be called when the user pushes a mouse button within the largest centered square of the window. Analogously, if **MoveMonitoring** is set, the window's **MouseMove** method will be called continuously as the mouse moves across the window. This behavior is useful if you want to be notified of mouse movement and button clicking. Within **MouseClick** and **MouseMove**, you can examine several **WindowObj** fields that give more detailed information on the state of the mouse: **ButtonDown** is TRUE if the mouse button is currently down; **Button** gives the number of the button that was pressed or released; **SecondClick** is True if the most recent click is the second click of a double-click; and **BackgroundClick** is TRUE unless the mouse was clicked down on a selectable object.

To monitor mouse clicks or mouse moves, define an object that is derived from **WindowObj**, and that overrides the **MouseMove** and **MouseClick** methods. The definition for this object would look something like this:

```
DEFINITION MODULE Example4;

    FROM Window IMPORT WindowObj;

 TYPE

   MyWindowObj = OBJECT(WindowObj);
    OVERRIDE
      ASK METHOD MouseMove (IN x, y : REAL);
      ASK METHOD MouseClick (IN mouseX, mouseY :
       REAL; IN buttonDown : BOOLEAN);
   END OBJECT;

 END MODULE.
```

. . . and the implementation module like this:

```
IMPLEMENTATION MODULE Example4;

OBJECT MyWindowObj;
  ASK METHOD MouseMove(IN x, y : REAL);
  BEGIN
    OUTPUT("The position of the mouse is: x = ", x,
     " y = ", y);
    INHERITED MouseMove(x, y);
      { this sets the window fields }
  END METHOD;

  ASK METHOD MouseClick(IN x, y : REAL; IN
    buttonDown : BOOLEAN);
  BEGIN
    OUTPUT("Mouse was clicked at: x = ", x,
     " y = ", y);
    OUTPUT("Mouse Button #", Button, " clicked");
    OUTPUT("The state of the button is: ",
     ORD(buttonDown), " 0=Up 1=Down");
    IF (SecondClick) OUTPUT("A SecondClick!");
    END IF;
    IF (BackgroundClick) OUTPUT("A BackgroundClick!");
    END IF;
    INHERITED MouseClick(x, y, buttonDown);
  END METHOD;
  END OBJECT;

  END MODULE.
```

A simple program that used these objects to report mouse movement and clicking to the user would look like:

```
MAIN MODULE MainEx4;

FROM GTypes   IMPORT WaitForEvent;
FROM GProcs   IMPORT HandleEvents;
FROM Example4 IMPORT MyWindowObj;

VAR       mywindow : MyWindowObj;

BEGIN
    { Create window with overridden mouse click and
```

```
        mouse move methods. }
    NEW(mywindow);
    ASK mywindow TO Draw;
      { Now wait in a loop handling user input }
    LOOP
       HandleEvents(WaitForEvent);
    END LOOP;
  END MODULE.
```

All mouse coordinates are specified using the world coordinate system given to the window with the **ShowWorld** method. If the world is not specified, then the default world is used (**0...32767**, **0...32767**).

4.7 Scroll Bars

WindowObj objects can contain both vertical and horizontal scroll bars. The size and position of the thumb contained in a scroll bar can be set programmatically. Thumb size is specified as a percentage of the total scroll bar size (as a **REAL** in the interval [0.0, 1.0]). The horizontal scroll position is specified as the distance of the leftmost side of the thumb from the left side of the scrollbar (as a **REAL** in **[0.0, 1.0-ThumbWidth]).** Vertical scroll position is the distance of the top side of the thumb from the top of the vertical scroll bar (as a **REAL** in **[0.0, 1.0-ThumbHeight]**). Whenever a thumb position is changed by the user, the **BeScrolled** method is called.

Note: Window contents are NOT scrolled automatically. This is the responsibility of the programmer.

4.8 Using the Status Bar on a Window

Windows can also have status bars. A status bar is a set of text output boxes or *panes* attached to the bottom of a window. The text in these panes can be set programmatically, but cannot be changed by a user. Panes are numbered from left to right starting with pane 0. The **SetNumPanes** method defines how many status bar panes to show. (No status bar will appear if the number of panes is zero.) The width (in character units) of each pane except pane 0 can be defined using the **SetPaneWidth** method. To set the text shown in a particular pane, use the **ShowStatus** method.

The leftmost status pane or *pane 0* is special for two reasons. First, it is used to display status bar messages shown when the pointer passes over a menu item. (See **MenuItemObj.SetMessage**.) In addition, the width of this pane varies with the width of the window. Therefore, the space for panes 1, 2, ..., n will not be truncated unless absolutely nessessary.

4.9 Asynchronous Notification of Window Close and Resize Events

Many toolkits provide a mechanism for closing windows while an application is running. Whenever a user closes a SIMGRAPHICS II window, that window's **BeClosed** method is automatically invoked. The default behavior of **BeClosed** is to terminate the application, but you can **OVERRIDE** this method to do something else.

EXAMPLE:

```
MyWindowObj = OBJECT(WindowObj)
 OVERRIDE
  ASK METHOD BeClosed;
END OBJECT:

OBJECT MyWindowObj;
 ASK METHOD BeClosed;
 BEGIN
   OUTPUT("Window has been closed by the user!");
   ASK window TO Erase;
     { don't quit, just get rid of window! }
 END METHOD;
END OBJECT;
```

Similarly, for resize events:

```
MyWindowObj = OBJECT(WindowObj)
 OVERRIDE
  ASK METHOD BeResized;
END OBJECT:

OBJECT MyWindowObj;
 ASK METHOD BeResized;
 BEGIN
   OUTPUT("Window has been resized by the user!");
 END METHOD;
END OBJECT;
```

Whenever you resize a window, that window's **BeResized** method is automatically invoked. You may override the **BeResized** method, but you should do an inherited **BeResized** for all of the windows contents to be rendered.

4.10 Printing the Contents of a Window

The system-specific print dialog and print mechanism allows individual SIMGRAPHICS II image trees as well as entire windows to be dumped to any installed printer. This is a big advantage under Microsoft Windows where the standard system print dialog allows printing to a wide variety of printers. For systems that do not have native printing facilities, a generic print dialog box is provided which generates Encapsulated Postscript files. The following method is defined in module **Graphic**:

```
ASK METHOD PrintGraphic (IN usedialog: BOOLEAN)
                        : BOOLEAN ;
```

This procedure prints the contents of a visible **WindowObj** or **ImageObj** in a system-specific manner. If **usedialog** is TRUE, a modal system print dialog box will appear (if appropriate for the platform) to allow printer selection, format options, etc. If FALSE, current defaults will be used.

Return values are TRUE for success and FALSE for failure.

4.10.1 Rules for System Printing

1. Only **WindowObj**'s or **ImageObj**s (or user-derived subclasses of them) can be printed. For instance, **ChartObj**s will not be printed if asked to **PrintGraphic**. However, it can be printed by attaching the chart to an image tree and asking for an **ImageObj** or **WindowObj** ancestor to **PrintGraphic**.
2. Any **ImageObj** to be printed should be attached to an image tree rooted at a **WindowObj** which has been asked to **Draw**; otherwise, certain internal scaling information will not be computed correctly. **WindowObj**'s asked to **PrintGraphic** should be visible.

4.11 Frame and Sub-windows

This feature is implemented for Windows NT and Windows 95 only. All windows can now be made sub-windows of an application frame window. One common 'frame' window will contain all sub-windows. The frame window is constructed automatically by the application when the first sub-window is displayed. These sub-windows can be tiled, cascaded or arranged as you desire, but they always remain within the main frame window of the application. (The **Window** menu which is displayed automatically provides these capabilities) When a sub-window is brought to the front, its palettes and menu bar will become part of the frame. In addition, the first and last menus of the first sub-window (usually the **File** and **Help** menus) will be shared among all other sub-windows.

The following procedures are available for manipulating the **frame window**:

```
PROCEDURE SetFrameTitle(IN title : STRING);
```

This procedure will reset the title displayed on the header bar of the frame

window. The procedure can be called before or after the frame window has been made visible, and will automatically update the title.

PROCEDURE SetFrameIconNames(IN smallIconName, largeIconName : STRING);

Sets the icons used when the frame window is minimized. If the application contains a frame and sub-windows, this procedure will identify either the resource or bitmap file names of the icons representing the minimized application.

PROCEDURE SetFrameTranslation(IN tx, ty : PctType)

If the application contains a frame and sub-windows, this procedure will specify the initial position of the lower left hand corner of the frame window. Position is specified in "screen" coordinates (where the lower left hand corner of the computer screen is (0,0) and the upper right corner is (100,100). This procedure must be called before the first sub-window is drawn.

PROCEDURE SetFrameSize(IN width, height : PctType)

If the application contains a frame and sub-windows, this procedure will specify the initial size in "screen" coordinates ([0,100], [0,100]) of the frame window. This procedure must be called before the first sub-window is drawn.

PROCEDURE GetFrameTranslation(OUT tx, ty : PctType)

If the application contains a frame and sub-windows, this procedure will retrieve the current position of the lower left hand corner of the frame window. Position is specified in "screen" coordinates [0,100], [0,100].

PROCEDURE GetFrameSize(OUT width, height : PctType)

Gets the current size of the frame window. If a sub-windowed style application is being used, this procedure will retrieve the current size in "screen" coordinates ([0,100], [0,100]) of the frame window.

Methods of **WindowObj** and **ControlWindowObj** relating to sub-windows:

ASK METHOD SetSubWindow(IN isSubWindow : BOOLEAN);

Enables the application frame / sub-window interaction. Instructs the window to be a sub-window inside a frame window. The frame window does not correspond to any particular **WindowObj** object, but is manipulated through procedures in the module **GProcs**.

ASK METHOD SetInitialState(IN position : WindowStateType);

Sets initial state (minimized, maximized) of the sub-window. **NormalWindowState** means the sub-window will appear somewhere within the frame window at less than maximized size. **MinimizedWindowState** means the window will appear as an icon. **MaximizedWindowState** means the window will appear maximized to the full extent of the frame window. Implemented for Windows NT and Windows 95 platforms only. Has no effect on other platforms.

ASK METHOD SetAutoGeometry(IN useDefaultSizeAndPosition : BOOLEAN);

Sets flag enabling the frame window to determine initial size and position of a sub window. IF TRUE, windowTranslation, Width, and Height fields are ignored.

ASK METHOD SetIconNames(IN sIconName, lIconName : STRING);

Sets the names of icon bitmaps shown when the window is minimized. Implemented for Windows NT and Windows 95 platforms only. Has no effect on other platforms.

ASK METHOD BeActivated;

This method is called when the you activate a sub-window. Override this method to be notified of when a sub-window is brought to the front.

4.12 Control Windows

SIMGRAPHICS II offers the ability to add controls like buttons and value boxes to windows rather than just dialogs. A *control window* is a window that contains any controls normally found inside a dialog box. Control windows are implemented via the **ControlWindowObj** object found in definition module **Window**. The following features apply to control windows:

1. Control windows are sized and positioned exactly like conventional windows u sing the **SetSize** and **SetTranslation** methods with the parameters given in "Screen Space" (0,0) to (100,100).
2. Unlike dialog boxes, you can resize a control window. You can also minimize and maximize a control window.
3. Control windows do not automatically grow to fit their contents like dialog boxes. A control window will automatically attach and manage scroll bars if there is not enough visible area to show all of its controls.
4. A control window can be managed as a sub-window inside a frame window.
5. The **ListBoxMultObj**, **TableObj**, **TreeObj** and **MultiLineBoxObj** controls can be maximized inside a control window. In this case, the control will occupy the full width and height of the canvas, and automatically resize with the

window. This allows you to have “table windows”, “tree windows”, and “text editor windows”.

Control windows handle their own mouse events. There is no notification of mouse movement or background clicking in a control window as there is with a standard window.

Chapter 5: Images

In MODSIM III, *Images* are shapes that are seen on the canvas of a window. The image provides many capabilities. The basic ones are:

- *Animation*: In MODSIM III, images are easily animated. When images in the same window pass over the top of each other they will be automatically re freshed by the runtime library.
- *Color*: Each **`ImageObj`** can have its own color. This color can either be defined by an enumerated constant of type **`ColorType,`** or be defined in terms of its RGB intensities.
- *Scaling*: The size of an image can be scaled in both x and y directions.
- *Rotation*: An image can be rotated a certain number of radians.
- *SnapShot*: For optimal animation performance, the library can optionally construct a snapshot for an image. This can be used to draw a complicated image with no performance penalty.
- *Hierarchically described*: An image can have subcomponents that are scaled, rotated and positioned with the whole image, but that can be manipulated individually.

5.1 Image Tree Used in Graphics Applications

In MODSIM III an image can be organized into a hierarchy using **`ImageObj`** objects. An **`ImageObj`** inherits the ability to contain graphical objects from **`GraphicVObj`**. Since objects contained in an **`ImageObj`** can also contain **`ImageObj`** objects, a hierarchy of images (or an image tree) can be constructed to represent the system that is to be graphically modeled. The images contained in an image are referred to as its children. The collection of the children, the childrens' children, etc., are referred to as an image's descendants. The container of an image is called its parent. The collection of an image's parent, its parent's parent, etc., is called its ancestors.

The ability to organize an image hierarchically is a powerful feature and can be used in a variety of ways. First of all, it can be used to break an object down into its subcomponents. Suppose you wanted to depict an image of a shopping cart. This cart could be broken down into a “body”, a “front wheel”, and a “back wheel.” You could use a hatched polygon for the body and circles for the wheels. The image tree for the cart would look something like this:

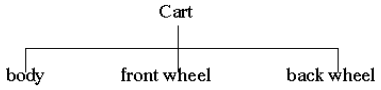


Figure 5-1. Image Tree for a Grocery Cart

Another way of using an image tree is for simple grouping. Suppose you had created a number of these carts. You could group them together into a common image called carts. Since any operation performed on an image is automatically performed on its children, this grouping would prove useful to you if you wanted to do something to all carts at once—like erase, move or scale them. Grouping can also be used as an organizational convenience. The image tree that groups the carts looks like this:

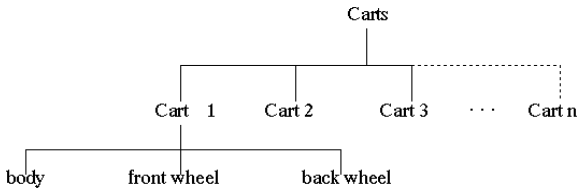


Figure 5-2. Grouping of Grocery Carts

The image tree is also used to describe the entire model. Suppose we were modeling a grocery store. Within the model we wanted to show shopping carts, checkout counters, and grocery aisles. The carts, counters and aisles would simply be children of their grocery store image.

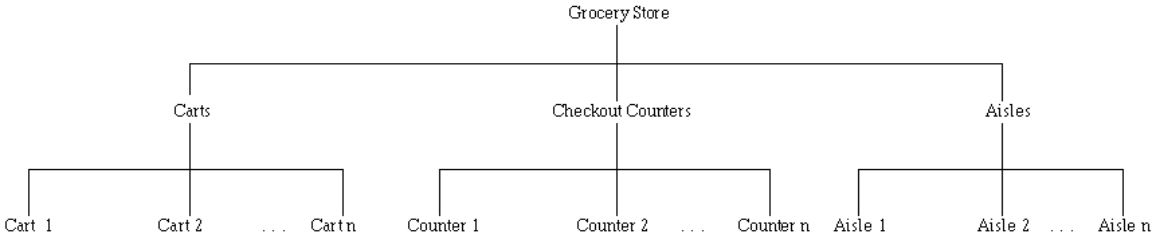


Figure 5-3. Image Tree for a Grocery Store

An image tree can be built within SIMDRAW using SIMDRAW's grouping facility, or can be constructed dynamically at runtime. The **Add** and **Remove** methods inherited from **GraphicVObj** are used to build and manipulate the tree structure. For example, suppose you wanted to graphically model the solar system. You could do this by creating an **ImageObj** instance for the entire solar system, and then add ten **ImageObj** instances to it that represented the planets and sun:

```
VAR
   solarSystem : ImageObj;
   sun, mercury, venus, earth, mars : ImageObj;

        { Get or create planet and sun images }

NEW(solarSystem);
ASK solarSystem TO AddGraphic(sun);
```

```
ASK solarSystem TO AddGraphic(mercury);
ASK solarSystem TO AddGraphic(venus);
ASK solarSystem TO AddGraphic(earth);
ASK solarSystem TO AddGraphic(mars);
...
```

When an operation is performed upon an image, the operation is automatically performed upon its children in the tree. In the example above, if you were to draw the solar system image, the planets would be drawn also. Erasing the solar system would cause all the planets to be erased. If you scaled down the solar system in size, the planets would automatically appear smaller, as well as having the physical distances between them shrink.

Position and sizing attributes of an image (such as scaling, rotation, and translation) are always set with respect to the image's parent in the image tree, not the window in which it lies. If an image rotation was set to 45 degrees, this means that the image would appear rotated 45 degrees from its parent in the tree. If the image's parent had also been rotated 45 degrees, then total rotation of the image seen in the window would be 90 degrees. In the above solar system example, suppose we wanted to move one of the planets. We would then specify a new position for the planet relative to the center of the solar system. Then, if we moved the solar system around the window, the planets would automatically move with it since their positioning relative to the solar system center has not changed.

5.2 Image Priority

Any image will appear on top of other images that come before it in its parent's list of children. It therefore appears beneath other images that come after it in the list. In the above block of code the Earth comes after Mercury and Venus in the solar system's list of children. Therefore, at times when the planets overlap, the earth would appear on top of Venus and Mercury. But since Earth comes BEFORE Mars in the list, Mars would appear on top of Earth during an overlap.

Figure 5-4. Image Priority

5.3 Creating and Using Images

Images can be created in one of two ways. One way is to construct them at runtime by creating instances and setting their display-specific attributes (such as points, color and style). This can be time consuming because there are many attributes that define what an image will look like. Also, the programmer cannot see what the images will look like on the screen until the program is compiled and run. For this reason the MODSIM III graphics package comes with SIMDRAW, an interactive, user friendly, graphics editor. Refer to chapter 2.

SIMDRAW can be used to construct simple primitives, or entire image trees. Within SIMDRAW you name the images and primitives you have constructed and would like to use in your program. SIMDRAW then creates a library of these objects and saves it to a file. As described in Chapter 4, a **`GraphicLibObj`** is used to obtain instances of the objects created in SIMDRAW:

```
{ ... Assume file "SolarSys.sg2" was created by
  SIMDRAW and contains the objects named
  "MERCURY", "VENUS", "EARTH" etc. ... }

VAR library : GraphicLibObj;
VAR mercury, venus, earth, mars : ImageObj;
...
NEW(library);
NEW(mercury);
```

```
NEW(venus);
NEW(earth);
NEW(mars);
ASK library TO ReadFromFile("SolarSys.sg2");
ASK mercury TO LoadFromLibrary(library,"MERCURY");
ASK venus TO LoadFromLibrary(library, "VENUS");
ASK earth TO LoadFromLibrary(library, "EARTH");
ASK mars TO LoadFromLibrary(library, "MARS");
...
```

The *grouping* facility of SIMDRAW is used to create images. When a set of images is grouped within SIMDRAW, an image is created for that set. Components of a group are optionally given **STRING** reference names and **INTEGER Id** tags. They are set in SIMDRAW using the **Edit/Properties** menu option. These are used by the application program to get handles to children of the image or grouping that SIMDRAW has created. The **GraphicVObj** ask method **Child** returns the child of the image that has this unique name and **INTEGER ID**. Suppose a solar system was created within . If the planet groupings seen in the editor were given the reference names Mercury, Venus, Earth, and Mars, the following block of code could be used to get handles to the planets of the solar system:

```
VAR library : GraphicLibObj;
VAR mercury, venus, earth, mars : ImageObj;
VAR solarSystem : ImageObj;
...
NEW(library);
NEW(solarSystem);
ASK library TO ReadFromFile("SolarSys.sg2");
ASK solarSystem TO LoadFromLibrary(library,
 "SOLAR SYSTEM");
mercury := ASK solarSystem Child("MERCURY", 0);
venus   := ASK solarSystem Child("VENUS", 0);
earth   := ASK solarSystem Child("EARTH", 0);
mars    := ASK solarSystem Child("MARS", 0);
...
```

Before any image can be drawn or updated, its image tree must be added to the window in which it is going to appear. The following code illustrates how to bring up a window and add an image to it:

```
VAR window : WindowObj;
VAR image  : ImageObj;
...
NEW(window);
```

```
  { create the window }
ASK window TO Draw();
  { bring it up on the screen }
ASK window TO AddGraphic(image);
  { add an image to it }
```

The following example gets a'solar system' from a library, and draws it on the screen:

```
MAIN MODULE Example2;

  FROM Window  IMPORT WindowObj;
  FROM GTypes  IMPORT WaitForEvent;
  FROM GProcs  IMPORT HandleEvents;
  FROM Graphic IMPORT GraphicLibObj;
  FROM Image   IMPORT ImageObj;

  VAR parentWindow : WindowObj;
  VAR solarSystem : ImageObj;
  VAR library : GraphicLibObj;

BEGIN
  NEW(parentWindow);
    { create a window }
  ASK parentWindow TO Draw();
    { bring it up on the screen }

  NEW(library);
    { create a library }
  ASK library TO ReadFromFile("Example2.sg2");
    { read the editor file }
  NEW(solarSystem);
  ASK solarSystem TO LoadFromLibrary(library,
  "SOLAR SYSTEM");
    { get a "SOLAR SYSTEM" from library }

  ASK parentWindow TO AddGraphic(solarSystem);
    { add the solar system to the window }

  ASK solarSystem TO Draw();
    { draw the entire solar system }
  LOOP
```

```
    { wait forever }
     HandleEvents(WaitForEvent);
  END LOOP;
END MODULE.
```

5.4 Coordinate Systems

When animating graphics, you must have a knowledge of the coordinate system in which the graphical objects live in order to specify their new positions. In MODSIM III, a coordinate system can be attached to any image, and the CHILDREN of the image live within this system. This means that all real world coordinates of an image's children must be specified in this coordinate system. Therefore, the position of any image in the image tree is always specified as the coordinates of its center point in PARENT coordinate system units. The center of any image is always point **(0.0, 0.0)**. Coordinate systems are specified for an image using the **SetWorld** method. If no world is set for an image, it will use its parent's world coordinate system. If none of the ancestors of an image contain a coordinate system, a default system is used. In this system **(WorldXlo, WorldYlo)** is the lower left-hand corner of the world, and **(WorldXhi, WorldYhi)** is the upper right-hand corner. (These constants are found in the library module **GTypes**.)

For example, suppose we wanted the positions of the planets within the solar system to be specified in terms of light minutes. We also want the solar system to include Mars as the furthest planet out, with the Sun as the solar system's center. The world boundaries would then be the distance of Mars from the Sun. This is illustrated by the following block of code:

```
...
ASK solarSystem TO SetWorld(-12.662, -12.662,
 12.662, 12.662);
ASK sun TO SetTranslation(0.0, 0.0);
  { sun is center of solar system }
ASK mercury TO SetTranslation(3.217, 0.0);
  { merc. dist. from sun }
ASK venus TO SetTranslation(6.011, 0.0);
  { venus dist. from sun }
ASK earth TO SetTranslation(8.310, 0.0);
  { earth dist. from sun }
ASK mars TO SetTranslation(12.662, 0.0);
  { mars dist. from sun }

{ ... assume no world is set for solar system
  parent --> use default world }
```

```
...
ASK solarSystem TO
 SetTranslation(WorldXlo / 2.0, WorldYlo / 2.0);
...
```

By default, the physical size of an image's world coordinate space is the largest centered square of the window in which the image lies. An important note is that when a transformation (scale, rotation, or translation) is applied to an image, its coordinate system's physical size and position with respect to the window is the ONLY thing that actually changes. This means that none of the real world points of an image actually change when it is scaled or rotated, but it physically changes size and position because its entire coordinate system has physically changed in size and position If the solar system image in the above example were to be scaled by (0.5, 0.5), the distance of the earth from the sun would remain at 8.31, but the solar system's world coordinate space would now only occupy one quarter the size of the window.

Figure 5-5. Solar System Coordinate System

5.5 Deriving from Images

You may derive new objects from **`ImageObj`**, however you should not use multiple inheritance to drive from the two graphical objects such as **`ImageObj`** and **`PolygonObj`** or **`ButtonObj`** and **`ImageObj`**. The most common case for deriving from **`ImageObj`** is to add additional fields and methods or to override the **`BeSelected`** method to obtain notification when the user has clicked on it.

5.6 Detecting Image Selection

Often it is useful to enable a user to make a selection between a number of images. This is done using the **AcceptInput** method and **LastPicked** field. Every graphical object has a **LastPicked** field. If **LastPicked <> NILOBJ**, it holds the highest descendant which is **Selectable**, which has a non-0 **Id** or non-null , and which was last clicked upon, or had a descendant last clicked upon. The method waits until such a descendant is clicked upon, and then returns **LastPicked**. As an example, assume **PanelImage** contains two objects with **"Truck"** and **"Car"**; assume further that **"Car"** contains two images, **"Body"** and **"Wheel"**. Then the following code waits for a mouse click on a descendant of **PanelImage** and determines exactly what was clicked upon:

```
selectedItem := ASK PanelImage TO AcceptInput();
  CASE ASK selectedItem ReferenceName
    WHEN "Truck"
    ...
    WHEN "Car" :
       selectedItem2 := ASK selectedItem LastPicked;
       CASE ASK selectedItem2 ReferenceName
        WHEN "Body":
          ...
        WHEN "Wheel":
          ...
       OTHERWISE
  END CASE;
```

If you need to detect image selection while a simulation is running you must override its **BeSelected** method and insert the code you want to execute when the image is selected. Within **BeSelected** you can use the methods **ClickButton()** and **ClickIsSecond()** to find out the button used to select, and whether or not this is a **Double Click**:

```
OBJECT Truck
  ASK METHOD BeSelected
  BEGIN
     ...
  END METHOD
END OBJECT.
```

The **BeSelected** method will be called automatically when the user clicks on the graphical object.

5.7 Getting Image Bounding Boxes

To aid in properly positioning and aligning images, the new **ImageObj** method can be used to obtain the bounding box in raw Normalized Display Coordinates (NDC). This is the same coordinate system used for the coordinates in the **WindowObj, MouseMove** and **MouseClick** callback methods. To convert these coordinates to a coordinate system defined by another image, use the **GProcs** module procedure "**Transform**", specifying **NILOBJ** for the "from" image:

```
ASK image TO GetBoundingBox(xlo, ylo, xhi, yhi);
Transform(NILOBJ, image.Parent, xlo,ylo, txlo, tylo);
Transform(NILOBJ, image.Parent, xli, yhi, txhi, tyhi);
```

5.8 Bitmapped Graphics

SIMGRAPHICS provides integrated vector and raster capabilities. Raster format files can be displayed as backgrounds, or icons, and combined with vector icons. The raster format supported is that of the one most commonly used on the target platform. On X Windows based workstations, the **xwd** format is used; on machines running Microsoft Windows, the .BMP format is used.

A **SnapShotObj** is used to display bitmaps. The bitmap to display is loaded from the file named in the **File** field of a **SnapShotObj**; more accurately, the bitmap is loaded from the file whose name is formed by concatenating **File** with a system dependent extension.

A **SnapShotObj** also supports (optional) bitmap masks, which are used for showing bitmaps that allow parts of the background to shine through. A mask bitmap is the same shape as the real bitmap, but contains only two colors: pixels colored with the minimum color index denote pixels where the background should show through, and all other pixels have the maximum color index. After loading a bitmap from **File**, the **SnapShotObj** looks for a file named **File**+"**m**"; if it finds one, its assumes that it contains a mask bitmap for the bitmap in **File**. The bitmap is not actually read in until it is asked to **Draw**.

The mask bitmap is used to define which areas of the bitmap should be displayed. If there is not a mask bitmap, the entire rectangle of the bitmap is used. Masks can be used to produce irregular shaped renderings of bitmaps, including holes.

A **SnapShotObj** can be defined as "Scalable". If it is scalable, it can grow and shrink like any **ImageObj**. Non-scalable bitmaps always appear at the same size and shape. A **SnapShotObj** never changes its appearance when asked to **Rotate**. **SnapShotObj** behaves like other primitives such as polygons, polylines, circles, etc., except that it cannot be rotated and it scales more slowly than the vector primitives. **SnapShotObj**'s may be added to image hierarchies along with vector primitives.

To create a bitmap that can be used by a **SnapShotObj**, consult the platform-specific release notes for the supported formats. Any scanning or painting package that can

generate an appropriate format may be used to create a bitmap file. Another possibility is to draw an object using SIMDRAW. SIMDRAW can create an appropriate bitmap file from any picture you can draw by selecting **File/Export Raster** menu option. A final possibility is to write a MODSIM program that creates an **ImageObj**; then the **WriteSnapShot()** method can be used to generate an appropriate bitmap file.

The following is an example of reading in a scalable bitmap called "coast" and displaying it. The bitmap is scaled to 1/2 the size of the window:

```
FROM GTypes IMPORT PointArrayType;
FROM GSnap IMPORT SnapShotObj;

PROCEDURE ReadBitmap(IN window : WindowObj);

VAR
  bitmap : SnapShotObj
  points : PointArrayType;

BEGIN
  ...

  NEW(bitmap);
  ASK window TO AddGraphic(bitmap);
  ASK bitmap TO SetFile("coast")'
  NEW(points, 0..1);
  points[0] .x := 0.0;
  points[0] .y := 0.0;
  points[1] .x := 16384.0; (* 1/2 the window size in
ndc's*)
  points[1] .y := 16384.0;
  ASK bitmap TO SetPoints(points);
  DISPOSE(points);
  ASK bitmap TO Draw;

  ...

END PROCEDURE;
```

5.8.1 Zooming into Bitmaps

You must be careful when scaling bitmaps, since they can easily consume large amounts of memory. This problem becomes apparent if you zoom into a section of a bitmap. The scaling operation could produce an enormous bitmap which would take a long time to render or would crash the program because of a lack of memory.

You may use the **SetPortion** method of the **SnapShotObj** to limit the area of the bitmap that is drawn. This is useful when zooming to prevent the entire bitmap from b eing

scaled.

5.8.2 Bitmap Alignment (Centering)

Normally, the position of a **SnapShotObj** object is specified with respect to its center point at the lower left hand corner. The center point can be changed through the **SetAlignment** method. For example, to center a non-scalable bitmap in the window (assuming you are using the "default" coordinate system), use the following code:

```
  . . .
ASK snapShot TO SetAlignment(SnapHorizCentered, SnapVertMiddle);
ASK snapShot TO SetTranslation(WorldXhi/2.0, WorldYhi/2.0);
  . . .
```

5.8.3 Converting Images into PostScript

Any **ImageObj** object can be converted to EPS PostScript using either SIMDRAW, or program code. **ImageObj** objects can be converted to PostScript using SIMDRAW. To do this, first save the object you wish to convert to a SIMGRAPHICS II library file using the **SaveToLibrary** and **WriteToFile** methods described in Chapter 3. Then load the image into the SIMDRAW **Image Editor** and use the **File/Export** menu option. The following options are available for conversion:

1. Use built-in PostScript fonts instead of SIMGRAPHICS II fonts.
2. Specify the number of copies to print.
3. Show or not show the background.
4. Set background color to RGB (values between 0.0 and 100.0).
5. Draw a window border and generate a title
6. Specify width and height of window in inches.
7. Position image window on paper in inches from the lower left of the page.
8. Convert bitmaps using color or greyscale pixel values.

A specific image or the entire contents of a window can be converted into PostScript using program code through the **EPSObj** object found the module **GEPS**. Use the **SetFile** method to set the name of the EPS file you wish to write to. Use other 'set' methods to set up conversion options. Pass either a **WindowObj** or **ImageObj** object to the **Convert** method to create the EPS postscript file. The following code could be used to perform conversion:

```
VAR
    eps : EPSObj;
    window : WindowObj;
...
```

```
NEW(eps);
ASK eps TO SetFile("tempfile.eps");
ASK eps TO Convert(window);
```

The **EPSObj** object provides the following **ASK** methods:

ASK METHOD SetFile(IN pathName : STRING);
Sets the name of the file which will be opened. If no file name is set, **OutputLine()** will do nothing.

ASK METHOD Convert(IN graphic : GraphicVObj)
Converts graphic into postscript. At this point the file is opened for writing (if **File** <> "") .

ASK METHOD OutputLine(IN pstring : STRING)
This method is called for each postscript line that is generated. The default behavior of this method is to write the line out to the file.

ASK METHOD SetCopies(IN PageCount : INTEGER)
Sets number of copies to print.

ASK METHOD SetSize(IN size : REAL)
Sets size of output copy in inches. It is always square. Equivalent to **SetSizes** (size, size).

ASK METHOD SetSizes(IN width, height : REAL)
Sets output area of page in inches. This is a new routine to support non-square window output and enable full page use.

ASK METHOD SetOrientation(IN orient : PSOrientationType)
Sets orientation of image on page. This is a new routine to support non-square window output and enables full page use. Landscape orientation will cause the X-axis of the image to be aligned along the long dimension of the paper (the same direction specified by the **SetSizes** height argument). Default is **Portrait**.

ASK METHOD SetSizeToFit(IN flag : BOOLEAN)
Allows image to be scaled to fit drawing area. Supports non-square window output and enables full page use. If set **TRUE**, the **ImageObj** or **WindowObj** graphics will be scaled (by the same amount in both x and y) to fit in the specified output area. See **SetSizes**.

ASK METHOD SetOffset(IN x, y : REAL)
Sets offset from lower left corner of page in inches.

Note: Width refers to the default x direction (narrow dimension of paper) REGARDLESS of the orientation of the imageSee **SetOrientation**.

ASK METHOD SetBackground(IN r, g, b : REAL)

Sets background color for the image that is printed. Notice that printed pages are usually white, while SG2 application windows rarely are. If the application's text was white, for instance, it would be invisible on a white page. Use this method to change the background to a contrasting color (such as black).

ASK METHOD SetShowWindow(IN show : BOOLEAN)

If **show = TRUE** displays a window around the printed image.

ASK METHOD SetWindowName(IN name : STRING)

Displays name at top of image if **SetShowWindow = TRUE**.

ASK METHOD SetPSFont(IN UseBuiltInFont : BOOLEAN)

If **SetFont = TRUE** substitutes PostScript fonts for stroked SG2 fonts. PostScript fonts are always substituted for System Text fonts.

ASK METHOD SetPSTarget(IN target : PSTargetType)

Specifies how **SnapShotObj**s are encoded into PostScript. Generates **SnapShotObj** Postscript for Level 1 black and white (**PSTGrey**), color/Level 2 (**PSTColor**), or portable format for both (**PSTBoth**). Default is **PSTBoth**. Note that the portable format will result in a larger file than the others; the black/white/grey format is the smallest.

Chapter 6: Dynamic Objects

MODSIM III provides for a connection between graphical objects and simulation. Objects that form this connection are called *dynamic objects*. All dynamic objects inherit the properties of **`ImageObj`** (which inherits properties of **`GraphicVObj`**). Some of the most useful inherited methods are summarized below:

`ASK METHOD Draw()`
Draws a graphical object.

`ASK METHOD Erase()`
Erases a graphical object.

`ASK METHOD DisplayAt(IN x,y  REAL)`
Displays a graphical object at (x,y).

`ASK METHOD LoadFromLibrary(IN library : GraphicLibObj; IN name : STRING)`
Loads the description of a graphical object from a library.

`ASK METHOD BeSelected()`
Automatically invoked when object is clicked on. Can be overridden to receive asynchronous selection.

`ASK METHOD AddGraphic(IN graphic : GraphicVObj)`
Adds an object to the end of the set of graphical objects.

`ASK METHOD RemoveThisGraphic(IN graphic : GraphicVObj)`
Removes a specific object from the set of graphical objects.

`ASK METHOD Descendant(IN refName : STRING; IN id : INTEGER)`
Returns the object in the set with reference name **`refName`** and I.D. **`id`**. Recursively searches through sets of objects in set.

Note: These are only some of the methods inherited by dynamic objects. Other methods are outlined in chapter 15.

6.1 DynamicObj

There are several objects which provide access to the built-in simulation features of

MODSIM III. These objects are called dynamic objects and are derived from a common object called a **`DynamicObj`**.

The three methods of all dynamic objects are **StartMotion**, **StopMotion** and **DynamicUpdate**. A **StartMotion** call will cause the object to be periodically updated by the simulation timing routine. The **StopMotion** method will stop the object from being updated. The **DynamicUpdate** method is called by the timing mechanism to update the state of the object. It takes as arguments the current simulation time, and the elapsed simulation time since the last call. You can override this method to perform some appropriate action.

6.2 DynImageObj

A **DynImageObj** object combines the functionality of an **ImageObj** with that of a **DynamicObj**. It may be moved, scaled and rotated with respect to simulation time. It inherits the properties of three other objects: **MovingObj**, **RotatingObj**, and **ScalingObj**. The functions of these objects (and hence a **DynImageObj**) are as follows:

MovingObj

ASK METHOD SetCourse(IN course : REAL);
Sets direction which the object will travel. The course is specified in radians measured clockwise from the positive x-axis of the world coordinate system.

ASK METHOD SetSpeed(IN speed : REAL);
Sets the speed of the object in world coordinate units per time unit.

TELL METHOD MoveTo(IN XDestination, YDestination : REAL);
Moves the object to a specific point. The method finishes when the object arrives at the destination. The object's speed should be set before invoking **MoveTo**.

TELL METHOD FollowPath(IN path : PointArrayType);
Moves the object along a path defined by the array of points. This method finishes when the object has arrived at the last point in the array. Use **Interrupt** to stop it from continuing.

RotatingObj

ASK METHOD SetRotationSpeed(IN rotationSpeed : REAL);
Sets the speed of rotation in radians per second. Negative values cause clockwise rotation.

TELL METHOD RotateTo(IN theta : REAL);
Waits for an object to rotate to **theta** radians using **RotationSpeed**. The rotation speed should be set before invoking this method.

ScalingObj

ASK METHOD SetScaleSpeed(IN scaleSpeed : REAL);
Sets the amount that is added to an objects scaling factor every time unit. For example, with a scale speed of 1.0, an object will become twice as large after 1 time unit, 3 times as large after 2 time units, etc.

TELL METHOD ScaleTo(IN xScale, yScale : REAL);
Waits for an object to scale to **'xScale, yScale'** using **scaleSpeed**. The scale speed should be set before invoking this method.

Note: DynamicObj, MovingObj, RotatingObj, and **ScalingObj** are not graphical objects and cannot be displayed. Use a "**DynImageObj**" object which inherits all functionality of these objects.

Animating a **DynImageObj** can be done in one of two ways. One way is to set the course and speed of the object and ask it to **StartMotion**. This will cause it to move with respect to these attribute settings indefinitely. Another way is to **TELL** or **WAIT FOR** the object to **MoveTo**, **ScaleTo** or **RotateTo** a destination. These methods elapse simulated time and finish when the object has arrived at its destination.

6.3 DynClockVObj

Objects derived from **DynClockVObj** (**DynAClockObj** and **DynDClockObj**) are used to display simulated time. As a default, elapsed simulation time is displayed as hours on the clock. The method **SetTimeScale** can be used to specify the number of clock hours per unit of simulated time.

A dynamic clock must be asked to **StartMotion()** before it will show simulation time. The following code sets up a dynamic clock that was named **Analog Clock** in the editor:

```
...
VAR
  dynclock : DynAClockObj;
  window : WindowObj;
  library : GraphicLibObj;
...
NEW(dynclock);
ASK dynclock TO LoadFromLibrary(library, "Analog Clock");
ASK window TO AddGraphic(dynclock);
ASK dynclock TO StartMotion;
...
StartSimulation;
...
```

6.4 Time Scaling

When using animated graphics within a simulation, simulation time elapsed in a **WAIT** statement must also elapse real time. The global variable **Timescale** found in module **SimMod** specifies the number of real seconds that will pass for each unit of simulation time. Its default value is 1.0.

Note: When the first graphical object is created, this variable is initialized to 1.0. Be sure to set **Timescale** *after* creating the first graphical object.

6.5 Example of a Small Graphical Simulation

The following example creates a clock to display simulation time, and moves a truck image to the center of the window using the **MoveTo** method:

```
MAIN MODULE Example6;

FROM Animate IMPORT DynImageObj, DynAClockObj;
FROM Window  IMPORT WindowObj;
FROM Graphic IMPORT GraphicLibObj;
FROM SimMod  IMPORT StartSimulation; Timescale;
VAR
   truck  : DynImageObj;
   clock  : DynAClockObj;
   window : WindowObj;
   lib    : GraphicLibObj;

BEGIN
     { create objects }
   NEW(window);
   NEW(lib);
   NEW(truck);
   NEW(clock);

    { load in defs. of objects }
   ASK lib   TO ReadFromFile("Example6.sg2");
   ASK truck TO LoadFromLibrary(lib, "Truck");
   ASK clock TO LoadFromLibrary(lib, "Clock");

    { add objects to the window }
   ASK window TO AddGraphic(truck);
   ASK window TO AddGraphic(clock);
   ASK window TO Draw;
```

```
    { start the clock's motion }
   ASK clock  TO StartMotion;

    { set speed of truck }
   ASK truck TO SetSpeed(1000.0);
    { move truck to window center }
   TELL truck TO MoveTo(16384.0, 16384.0);
    { two seconds for every time unit }
   Timescale := 2.0;
    { start animation }
   StartSimulation;

END MODULE.
```

6.6 Deriving Objects from DynImageObj

There is a general problem of allowing the methods of an **ImageObj** to be overridden when the Image has been created by SIMDRAW. You may need to create your own object derived from some graphical object, but you need your new object to have all of the information set in SIMDRAW. This is accomplished using the **Associate()** method. The **Associate()** method copies all of the data set by SIMDRAW into the user defined object. It will also automatically put your object into the image tree at the time a **LoadFromLibrary** is performed.

```
TYPE
  TurretObj = OBJECT(DynImageObj)
   TELL METHOD PointTo(IN angle : DegreeType);
  END OBJECT;

  TankObj = OBJECT(DynImageObj)
   TELL METHOD Fire;
  END OBJECT;

{ Assume the turret made in SIMDRAW has reference name
"Turret" and id 0,  Also assume it is a subgouping
attached to a "TANK" }

VAR
  turret : TurretObj;
  tank : TankObj;
  ...
```

```
    NEW(tank);
    NEW(turret);
    ASK tank TO Associate(turret, "Turret", 0);
    ASK tank TO LoadFromLibrary(library, "TANK");

  { My 'TurretObj' object will now be automatically put
  into the image tree, and have all the graphical
  attributes set up in the editor. }
```

This is the mechanism used to substitute user defined images for images created by SIMDRAW. All of the attributes and child images of the old image are copied to the new image, and then the old image is disposed of.

Chapter 7: Graphs

In SIMGRAPHICS II, graph objects are used to graphically display a dynamically changing value or set of values. The generic **`GraphVObj`** object is used to refer to a graph object. There are a number of objects derived from **`GraphVObj`** and a variety of ways that data can be displayed. A set of data can be displayed using a piechart or a 2-D chart. Single values can be displayed using a dial, level meter or digital display, and time can be shown with a clock. A **`GraphVObj`** has the following properties:

- *Construction*—A graph is built using SIMDRAW.
- *Automatic rescaling*—2-D graphs, dials and level meters are rescaled automatically if one of the values in its data sets exceeds its range.
- *Easy updating*—If a new value is set in a graph, the **`Draw`** method will do whatever is necessary to redisplay this value. Because all graphs are derived from **`ImageObj`**, any operation that can be performed on an image can also be performed on a graph.

7.1 Objects Derived from GraphVObj

These are the presentation graphics objects:

`GraphVObj` — Generic object meaning any graph type.

`ChartObj` — A chart containing a data plot, legends, and titling.

`ClockVObj` — A generic object describing a clock.

`AnalogClockObj` — A graphical display of an analog clock.

`DigitalClockObj` — A graphical display of a digital clock.

`PiechartObj` — A piechart containing slices, legends, and titling.

`MeterVObj` — A generic object for any graphical display of a single value of type REAL.

`DialObj` — A graphical display of an analog dial.

`DigitalDisplayObj` — A simple display of a single value.

`LevelMeterObj` — A graphical display of a level meter.

`TextDisplayObj` — A graphical display of a text string.

The class inheritance tree for MODSIM III presentation graphics looks like this:

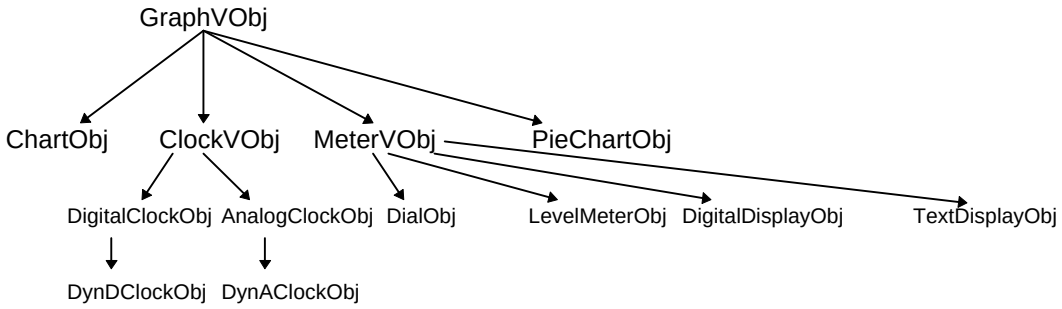


Figure 7-1. Inheritance Tree for Presentation Graphics

7.2 Creating and Using Graphs

The above types of graphs are created using SIMDRAW. Since a **GraphVObj** is inherited from **ImageObj**, all operations that can be performed upon an image can also be performed on a graph. Therefore, loading a graph from a library is done in the same manner as any image. For example, if you wanted to get a dial that was given the name "**Fuel Gauge**" in SIMDRAW, the following code would be used:

```
VAR library : GraphicLibObj;
VAR fuelGauge : DialObj;
...
NEW(library);
ASK library TO ReadFromFile("graphs.sg2");
NEW(fuelGauge);
ASK fuelGauge TO LoadFromLibrary(library,
 "Fuel Gauge");
...
```

If not set within SIMDRAW, the size of a graph can be set using the **SetViewbox** ask method. Its arguments are width and height specified in parent coordinate space units. To set the size of a dial to 10000.0 units wide by 10000.0 units tall:

```
...
ASK dial TO SetViewbox(10000.0, 10000.0);
...
```

The position of the graph is set within SIMDRAW, but can be set at runtime in the same manner that images are positioned, using the **SetTranslation** method:

```
...
ASK fuelGauge TO SetTranslation(x,y);
...
```

SetTranslation sets the upper left corner for all graph types.

7.3 Description of Various Graph Objects

7.3.1 ChartObj

A chart is used to present a 2-D plot containing a number of data sets. It has a title for itself, and for each of the axes of its 2-D plot. It also (optionally) contains legends for each of the data sets within its 2-D plot. Within SIMDRAW you can specify whether the 2-D plot is a trace plot or a simple plot. You also specify how each of the data sets is to be represented.

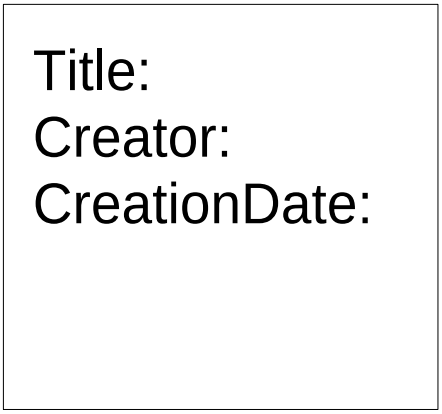


Figure 7-2. 2-D Plot

The **Plot** method is used to plot a point within one of the data sets in the chart. The number of the data set to plot and the x and y coordinates are given to the method. If the Y value plotted extends beyond the Y-axis boundaries, that axis is automatically rescaled. The plot's X-axis is rescaled only if that plot is a trace plot. For example, to see y = 50.0 plotted at x = 20.0 within data set '1' the following code is used:

```
...
ASK chart TO Plot(1, 20.0, 50.0);
...
```

A chart can also be plotted with the **SetCoordinate** method. Using this method, the change to the graph is not made visually apparent until it is asked to **Draw**. Therefore multiple points can be plotted without seeing the effect until **Draw** is done.

A chart can optionally contain legends for each of the datasets. The name used in the legend is the name given to the data set within SIMDRAW.

The data sets contained in the 2-D plot can be represented as follows:

Histogram—Used to plot a fixed number of values. A plotted value or data cell is shown using a rectangle. Each rectangle spans across adjacent X-axis tick marks. Therefore, the width of each data cell is exactly the X-axis tick mark interval.

Bar chart—This data set also contains a fixed number of cells. Each bar is a rectangle centered directly over an X-axis tick mark. When a coordinate is plotted, the bar nearest to the given x coordinate is modified to reflect the magnitude of the given y coordinate.

Discrete surface—A surface chart data set contains a fixed number of cells. A series of polygons or lines is used to show the entire data set as a surface. Each point is plotted directly over each x-axis tick mark.

Continuous Surface—A simple plot has a variable number of cells. Data is always added to this data set; in other words data cells are never replotted as in the above representations. This representation can be useful in post-processing a large amount of data when every piece of the data must be seen somewhere in the plot. Data sets shown in a trace plot are always represented in this fashion.

Plot **`stacking`** is useful when multiple sets with approximately the same values are plotted, where they would tend to obscure each other, or when it's desirable to present each data set value in relation to the sum of all data set values for a given cell in the manner of a pie chart. Fixed width data sets with the same representation can be displayed in stacked form, where the values plotted for each data set are added to the sum of the previously plotted values. The data sets are plotted from the bottom up, in order of increasing data set number.

For example, suppose we have two data sets plotted using a bar chart representation, whose first cell values are 1 and 3, respectively. If **`stacking`** is off, the second data set bar would cover the first data set bar completely. With **`stacking`** on, the first data set value is represented as a bar extending from 0 to 1 and the second data set value is plotted as a bar extending above the first from 1 to 4.

Data sets of the histogram and bar chart variety can be displayed side by side. In this case, individual bars and rectangles are made thin enough to allow room for bars and rectangles from the other datasets. Therefore, all data sets of the same representation can be seen within a single cell. Laying data sets side by side can be useful because no data sets are obscured.

A value plotted in a data set can be retrieved using the **`ChartObj`** method **`Y(datasetnum, x)`**. Given a data set number and an x coordinate, the y value plotted at that coordinate is returned. If the representation of the data set is of the *simple plot* variety, linear interpolation is used to get the exact y value shown at the given x value. Suppose a user wanted to know what was plotted at x=20.0 in data set 2:

```
VAR yval : REAL;
...
yval := ASK chart Y(2, 20.0);
...
```

Another property that charts have is the ability to label key parts of their 2-D plot. Labels will automatically move with the rest of the plotted data when the graph is rescaled. Labeling is done using the **PlotLabel** method. This takes an x,y coordinate and an object as the label. For example, suppose you want to label the point (20.0, 30.0) with the string **"<-- Peak Value"**. You also want this label to be rotated 45 degrees and the **"<"** character in the label to rest over the (20.0, 30.0) point. The following code would accomplish this:

```
VAR label : TextObj;
VAR chart : ChartObj;
...
NEW(label);
ASK label TO SetText("<— Peak Value");
ASK label TO SetAlignment(HorizLeft, VertMiddle);
ASK label TO SetRotation(pi / 4.0);
ASK chart TO PlotLabel(label, 20.0, 30.0);
```

Occasionally, it may be necessary to change some of the attributes of a **ChartObj** (e.g. Intervals, X-axis minimum, Y-axis minimum) to fit the set of data that will be plotted. There are several methods of **ChartObj** that can do this:

ASK METHOD SetRanges(IN xmin, xmax, ymin, ymax : REAL);
Sets the maximum and minimum values shown on the ends of the axes of a **ChartObj.**

ASK METHOD SetIntervals(IN xinterval, yinterval : REAL);
Sets the X-axis and Y-axis tick mark intervals. Also sets the X-axis bar width interval.

ASK METHOD SetNumIntervals(IN xnuminterval, ynuminterval : REAL);
Sets the X-axis and Y-axis numbering intervals.

ASK METHOD SetGridIntervals(IN xgridinterval, ygridinterval : REAL);
Sets the X-axis and Y-axis grid line intervals.

ASK METHOD SetIntercepts(IN xintercept, yintercept : REAL);
Sets the axis intercepts. **xintercept** is the point along the X-axis where the Y-axis crosses over. **yintercept** is the point where the X-axis crosses over.

Note: The above methods should only be called AFTER a chart has been loaded in from a graphic library (e.g. the **`LoadFromLibrary`** method). Refer to paragraph 7.7 for more information concerning programmatic setting of graph attributes.

7.3.2 PiechartObj

A **`PiechartObj`** is a graph containing a piechart, a title, and legends for the pie slices. Each slice in the pie shows a value as a percentage of the sum of all the values.

Figure 7-3. Pie Chart

The slices in a pie chart are numbered starting with '1'. The numbering of slices begins at the 3:00 position with respect to the pie, and continues in a counter-clockwise direction. The **`DisplaySlice`** method is used to set the value of a pie slice given its number. The percentage value of the slice is automatically computed and displayed in the legend for the pie slice. For example, to change the value of slice number "**`2`**" to '**`57.0`**':

```
...
ASK piechart TO DisplaySlice(2, 57.0);
...
```

7.3.3 ClockVObj

A **`ClockVObj`** is a generic object describing a clock. Clocks are either analog or digital. Time is specified in hours, minutes, and seconds. The time value seen within a clock is bounded by the attributes **`MaxHours`**, **`MinutesPerHour`** and **`SecondsPerMinute`** (set within SIMDRAW). You can, however, set the clock to a time value that goes past one or more of these boundaries. If you do, your value will be rolled over to the next highest place. For example, if you specified a time value of 120 seconds to a clock with the **`SecondsPerMinute`** field set to 60, the clock would display 2:00. The time value displayed in a clock is specified by the **`DisplayTime`** method:

```
...
ASK clock TO DisplayTime(6, 45, 30);
   { set time to 6:45:30 (h:m:s) }
...
```

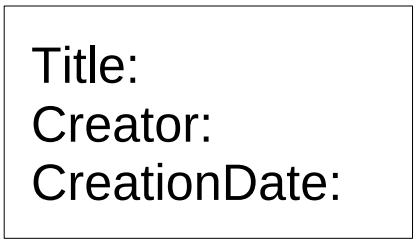


Figure 7-4. Digital Clock

Figure 7-5. Analog Clock

A **DigitalClockObj** is a graphical display of a digital clock. It is shown by a box containing a readout of the time displayed in “hours:minutes:seconds” format.

An **AnalogClockObj** is a graphical display of an analog clock. Like a conventional clock it has hour, minute and second hands.

If you are using the clock with a simulation you can use a **DynAClockObj** or **DynDClockObj** instead. These dynamic clock objects are automatically updated as simulation time advances. See paragraph 6.3 for more details.

7.3.4 MeterVObj

A **MeterVObj** is a generic object used to represent a single real value. A meter can be a dial, level meter, or digital display. All meters have a **Max** and **Min** field that specify their range. They also have a **ScaleFactor** (set within SIMDRAW). The value seen in the meter is actually the meter's value (set by the user) multiplied by its scale factor. This value is set using the **DisplayValue** method. Suppose you wanted to graphically display the value 100.0:

```
VAR meter : MeterVObj;
     ...
     ASK meter TO DisplayValue(100.0);
     ...
```

MeterVObjs have a method called **SetRange**, which sets the **Min** and **Max** fields for the meter.

One of the meter objects is a **DialObj**; this is a graphical display of a dial. It can have numbering on the inside or outside.

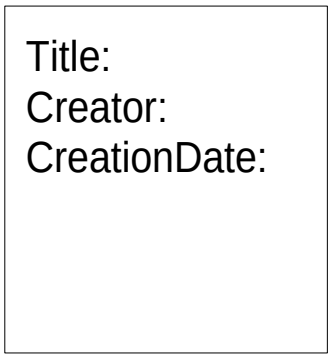


Figure 7-6. Dial

Figure 7-7. Level Meter

DialObjs have methods **SetBounds**, **SetNumOutside**, and **SetRange**. **SetBounds** sets the position in degrees of the maximum and minimum values on the dial. **MinTheta** and **MaxTheta** are integer fields that specify the minimum and maximum angular positions given in degrees.

SetNumOutside sets whether numbering will be outside the dial. **NumOutside** is a Boolean field that is **TRUE** if numbering is outside the face of the dial, and false, otherwise. The default is**FALSE**.

SetRange is inherited from **MeterVObj**.

Another meter is a **LevelMeterObj**. A level meter is a small rectangular 1-D graph containing a bar whose height represents the value being displayed.

LevelMeterObj has a method **SetGridLines**, which turns grid lines on and off. If the Boolean variable **Gridlines** is **TRUE**, grid lines are turned on, otherwise they are off.

SetRange is inherited from **MeterVObj**.

A **DigitalDisplayObj** is another meter type. This is a simple box with a title containing the value. The **FieldWidth** attribute of the digital display specifies the total number of spaces that the value will occupy (including decimal point and fractional part). The **Precision** attribute specifies the number of decimal places displayed. **SetFieldWidth** sets the number of spaces in the box for the value. **SetPrecision** sets the number of spaces allowed for any fractional part.

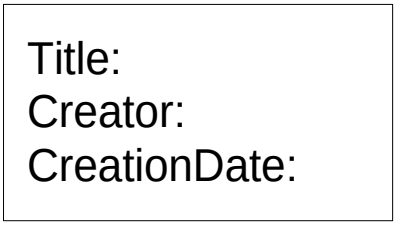


Figure 7-8. Digital Display

7.3.5 TextDisplayObj

A **TextDisplayObj** contains a graphical display of a value of type **STRING**. It has a title, and is composed of a simple box containing the string. The **field width** of a text meter is set within SIMDRAW, and if the length of the string is greater than this field width, then the string is truncated. This string is always aligned to the right end of the box. It can also be set with the method **SetFieldWidth**. The contents of a text meter are set by the **DisplayText** method. For example:

```
VAR textMeter : TextDisplayObj;
...
ASK textMeter TO DisplayText("Waiting For Input");
...
```

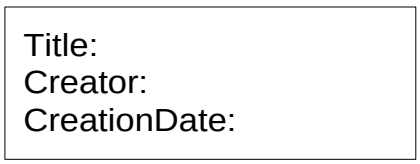


Figure 7-9. Text Display

7.4 Using Presentation Graphics to Monitor Variables

In MODSIM III a single variable or array of variables can be monitored using a SIMGRAPHICS II presentation graph. A single variable can be monitored by a level meter, dial, digital display, trace plot, chart, or pie chart. An array of variables can be monitored by a chart or pie chart. **STRING** variables can be monitored by a text display.

Refer to the *MODSIM III Reference Manual* for detailed information on variable monitors.

7.4.1 Single Variable Monitoring

Meters

A **data point** is used to monitor a single changing variable. There are three monitor objects provided by the SIMGRAPHICS II runtime library for data points. **IDataPtMObj** monitors an **INTEGER**, **RDataPtMObj** monitors a **REAL**, and **SDataPtMObj** monitors a **STRING** variable. A variable must be defined as being left monitored by one of these objects before its value can be shown on a graph. The **SetGraph** method of these objects makes the association between a SIMGRAPHICS II graph and the monitor object. The following code loads in a dial from a graphic library, and uses it to show the monitored variable **fuelRemaining**:

```
 VAR
  fuelRemaining : LMONITORED REAL BY RDataPtMObj;
  dial          : DialObj;
...
NEW(dial);
ASK dial TO LoadFromLibrary(graphicLib, "dial");
ASK window TO AddGraphic(dial);
...
ASK GETMONITOR(fuelRemaining, RDataPtMObj) TO
 SetGraph(dial);
...
fuelRemaining := 12.0;
```

Histograms

A data point can also be monitored using a histogram. In this case a **ChartObj** object must be used for the graph. This chart should be set up from the editor with the representation **Histogram, Bar Chart**, or **Surface Chart**. In addition, the **SetHistMode** method of the monitor object should be invoked with the parameter **TRUE**. The following code monitors the variable **queueLength** with a histogram:

```
VAR
  queueLength : LMONITORED INTEGER BY IDataPtMObj;
```

```
  histogram   : ChartObj;
...
NEW(histogram);
ASK histogram TO LoadFromLibrary(graphicLib,
  "Queue Length");
...
ASK GETMONITOR(queueLength, IDataPtMObj) TO
  SetGraph(histogram);
ASK GETMONITOR(queueLength, IDataPtMObj) TO
  SetHistMode(TRUE);
...
queueLength := 12;
```

Trace Plots

A single variable can be monitored with respect to time using a **Trace** plot. The **Trace** plot is shown with a **ChartObj** object, which must be set up as a **Time Trace Plot** in SIMDRAW. The monitoring is set up in the same fashion as dials, level meters and digital displays. When the value of the monitored variable changes, a point is plotted to the chart. The variable's new value is the y coordinate of the point and simulation time is the x coordinate, thus producing a graph of the variable over time.

Figure 7-10. Trace Plot

7.4.2 Showing More Than One Variable in the Same Chart

In SIMGRAPHICS II, two or more monitored variables can be shown in the same chart. This is accomplished with the **SetDataSet** method of the monitor object. This method is given the data set number the variable is to be shown in. For example, to show the

monitored variables **Position**, **Velocity** and **Acceleration** in the same chart, the following code would be used:

```
VAR
  Position, Velocity, Acceleration : RDataPt;
  tracePlot : ChartObj;
...
ASK GETMONITOR(Position, RDataPtMObj) TO
 SetGraph(tracePlot);
ASK GETMONITOR(Velocity, RDataPtMObj) TO
 SetGraph(tracePlot);
ASK GETMONITOR(Acceleration, RDataPtMObj) TO
 SetGraph(tracePlot);
ASK GETMONITOR(Position, RDataPtMObj) TO
 SetDataSet(1);
ASK GETMONITOR(Velocity, RDataPtMObj) TO
 SetDataSet(2);
ASK GETMONITOR(Acceleration, RDataPtMObj) TO
 SetDataSet(3);
...
```

A single variable can also be shown as the bar on a chart, or a slice on a pie chart. The **SetElement** method takes the bar or slice number of the chart that the data point is to be shown in. Bars on a chart are numbered from 1 to N going left to right. Slices on a pie chart are numbered from 1 to N going counter-clockwise starting at the 3:00 position in the pie. The following code would be used to show the variables **bostonSales**, **detroitSales** and **chicagoSales** in a piechart:

```
VAR
  bostonSales, detroitSales, chicagoSales :
    RDataPt;
  piechart : PiechartObj;
...
ASK GETMONITOR(bostonSales, RDataPtMObj) TO
 SetGraph(piechart);
ASK GETMONITOR(detroitSales, RDataPtMObj) TO
 SetGraph(piechart);
ASK GETMONITOR(chicagoSales, RDataPtMObj) TO
 SetGraph(piechart);
ASK GETMONITOR(bostonSales, RDataPtMObj) TO
 SetElement(1);
ASK GETMONITOR(detroitSales, RDataPtMObj) TO
```

```
 SetElement(2);
ASK GETMONITOR(chicagoSales, RDataPtMObj) TO
 SetElement(3);
...
```

Variables of type **STRING** can be shown using a **TextDisplayObj** object. The variable must be defined as being monitored by an **SDataPtMObj**. Setting up the monitoring is the same as with other types of variables.

7.4.3 Showing Arrays of Variables Using Charts

An array of variables can be shown using a chart or pie chart. The array type must be of type **LMONITORED RDataSet BY RDataSetMObj** for **REAL** arrays and of type **LMONITORED IDataSet BY IDataSetMObj** for **INTEGER** arrays. Elements in the array conform to either bars in a chart, or slices in a pie chart. The order of the array element determines what slice or bar will show its value. The first element in the array is shown by the first bar or slice in the chart, the second element by the second bar, etc. To show the array **valueArray** with a chart, the following code could be used:

```
From Graph IMPORT RDataSet, RDataSetMObj;

VAR
  valueArray : LMONITORED RDataSet BY
               RDataSetMObj;
  chart : ChartObj;
...
NEW(valueArray, 1..10);
ASK GETMONITOR(valueArray, RDataSetMObj) TO
 SetGraph(chart);
...
valueArray[3] := 27.0; {automatically updates
                        chart}
```

Note: Definitions for **DataPtMObj** objects are found in the module '**Graph**'. There are also several predefined types provided in this module that may be useful in defining monitored variables.

7.5 Graph Monitoring Table

MONITORING NEEDS	GRAPH OBJECTS	MONITOR OBJECTS	MONITOR METHODS
Single variable of type **INTEGER** or **REAL**	**DigitalDisplayObj** **DialObj** **LevelMeterObj**	**RDataPtMObj** **IDataPtMObj**	**SetGraph**
Single variable of type **STRING**	**TextDisplayObj**	**SDataPtMObj**	**SetGraph**
Histogram of a single variable	**ChartObj**	**RDataPtMObj** **IDataPtMObj**	**SetGraph** **SetHistMode**
Single variable plotted over time	**ChartObj** (trace plot in editor)	**RDataPtMObj** **IDataPtMObj**	**SetGraph**
Many single variables in the same graph (each shown as a bar or pie slice)	**ChartObj** **PiechartObj**	**RDataPtMObj** **IDataPtMObj**	**SetGraph** **SetElement**
Histograms of 2 or more variables in the same graph	**ChartObj**	**RDataPtMObj** **IDataPtMObj**	**SetGrap** **SetHistMode** **SetDataSet**
2 or more variables plotted over time in the same graph	**ChartObj** (trace plot in editor)	**RDataPtMObj** **IDataPtMObj**	**SetGraph** **SetDataSet**
2 or more sets of many single variables in the same graph (each single variable shown as a bar in a chart)	**ChartObj**	**RDataPtMObj** **IDataPtMObj**	**SetGraph** **SetDataSet** **SetElement**
Array of variables of type **REAL** or **INTEGER**	**ChartObj** **PiechartObj**	**RDataSetMObj** **IDataSetMObj**	**SetGraph**
2 or more arrays of variables of type **REAL** or **INTEGER** on the same graph	**ChartObj**	**RDataSetMObj** **IDataSetMObj**	**SetGraph** **SetDataSet**

7.6 Graph Example

Here is an example program using a chart

```
MAIN MODULE Example5;

{ This small program loads in a chart from a
  library, and plots a curve within it.

  We assume that the editor has created a chart
  named "Chart".  This graph should have it's XMin
```

```
  at -2.0, XMax at 2.0, YMin at -8.0, and YMax at
  8.0, and have 2 data sets.  These data sets
  should be of the 'simple plot' representation.
  It should be saved in "Example5.sg2" }
FROM Window IMPORT WindowObj;
FROM Graphic IMPORT GraphicLibObj;
FROM Chart IMPORT ChartObj;
FROM GProcs IMPORT HandleEvents;
VAR
    window : WindowObj
    library : GraphicLibObj;
    chart : ChartObj;
    x : REAL;
BEGIN
     { create a window and open it up }
   NEW(window);
   ASK window TO Draw();
     { read a "Example5.sg2" into a library, and
       get a chart called "Chart" }
   NEW(library);
   ASK library TO ReadFromFile("Example5.sg2");
   NEW(chart);
   ASK chart TO LoadFromLibrary(library, "Chart");
     { Add the chart to the window }
   ASK window TO AddGraphic(chart);
     { Plot the curve y = x^2  in dataset 1 and y =
       x^3 in data set 2 }
   x := -2.0;
   WHILE x <= 2.0
     ASK chart TO SetCoordinate(1, x, x*x);
     ASK chart TO SetCoordinate(2, x, x*x*x);
     x := x + 0.1;
   END WHILE;
     { Now 'see' the chart and the results of the
       plotting }
   ASK chart TO Draw();
     { wait }
   LOOP
     HandleEvents(TRUE);
   END LOOP;
 END MODULE.
```

7.7 Creating Graphs at Runtime

Using a set of provided **ASK** methods, a programmer can create any graph that can be created by SIMDRAW. Fill styles, line styles, mark styles, colors, and fonts of graph components can be specified. These components are identified in the enumerated type **GraphPartType** found in the **GTypes** module. The **ChartDataSet** and **PieSlice** parts must be used in conjunction with an integer identifying which data set component to modify. The following methods can be used to program the appearance of a graph.

7.7.1 Methods to Set the Color of a Graph Component

```
ASK METHOD SetPartColor
    (IN part : GraphPartType;  IN dsid : INTEGER;  IN
    color : ColorType);
ASK METHOD SetPartRGBColor
    (IN part : GraphPartType; IN dsid : INTEGER; IN r,g,b
    : PctType);
```

7.7.2 Methods to Set the Fill, Line, or Mark Styles of a Graph Component

```
ASK METHOD SetPartFillStyle
    (IN part : GraphPartType; IN dsid : INTEGER; IN fs :
    FillStyleType);

ASK METHOD SetPartLineStyle
    (IN part : GraphPartType; IN dsid : INTEGER; IN ls :
    LineStyleType;
     IN linePctWidth {range: 0 to 1} : REAL);

ASK METHOD SetPartMarkStyle
    (IN part : GraphPartType; IN dsid : INTEGER; IN ms :
    MarkStyleType);
```

7.7.3 Methods to Set the Text Fonts of Graph Components

```
ASK METHOD SetPartTextFont
    (IN part : GraphPartType; IN dsid : INTEGER; IN font
    : TextFontType);

ASK METHOD SetPartTextSysFont
    (IN part : GraphPartType; IN dsid : INTEGER;
     IN family : STRING; IN ptSize, weight, slant :
      INTEGER);
```

7.7.4 Method to 'Hide' a Graph Component

```
ASK METHOD SetPartHidden
    (IN part : GraphPartType; IN dsid : INTEGER;
    hiddenFlag : BOOLEAN);
```

7.7.5 Additional Methods for Programmatic Creation of a ClockVObj

The following methods are available for programmatic creation of a Clock:

ASK METHOD SetHandShowing
(IN showHours, showMinutes, showSeconds : BOOLEAN);

Controls the display of the hour, minute, and second indicators. Default is **TRUE** for displaying each indicator.

ASK METHOD SetHandScaling
(IN hoursPerDay, minutesPerHour, secondsPerMinute : REAL);

Sets the number of hours in a day, the number of minutes in an hour, and the number of seconds in a minute. The default values are 24, 60, and 60, respectively.

ASK METHOD SetMaxHours (IN maxShown : REAL);

Sets the highest hour displayed on the clock (i.e. the number at the top of an analog clock). The default value is 12.

ASK METHOD SetNumInterval (IN numHoursBetweenLabel : REAL);
(**AnalogClockObj** only)

Sets the number of hours between consecutive numbering. The default value is 3.

ASK METHOD SetInterval (IN ticInterval : REAL);
(**AnalogClockObj** only)

Sets the tick marking interval in hours.

7.7.6 PieChartObj Methods

The following methods enable programmatic creation of a pie chart:

ASK METHOD AddSlice;

Adds a slice to the pie chart. The new slice will be the last slice in the order.

ASK METHOD RemoveSlice (IN sliceNumber : SliceRefType);

Removes a pie slice from the chart given its number (order). High numbered slices will be consecutively renumbered.

```
ASK METHOD SetSliceTitle
    (IN sliceNumber : SliceRefType; IN sliceLegendText :
    STRING);
```

Sets the text shown in the legend for a particular pie slice.

7.7.7 Methods for Programmatic Creation of a ChartObj

Generally speaking, charts are defined through the **SetAxisField**, **SetOption**, and the **SetDSOption.** These methods accept enumerated type constants specifying which particular field or option of the chart is to be set.

Data sets are added and removed with the **AddDataSet** and **RemoveDataSet** methods. As before, a data set is specified by its ordinal number within the chart (an integer ranging from 1 to the total number of data sets).

7.7.8 Setting Chart Options

The **SetOption** method given below turns on or off an option for the chart. The options available are described by the **ChartOptionType** enumeration.

```
ASK METHOD SetOption(
    IN option : ChartOptionType; (* possible values
    listed below *)
    IN optionOn : BOOLEAN);
```

ChartShowLegend
If **TRUE,** chart will show a legend below the plot area. Calling the **SetDSTitle** method will set the data set description appearing in the legend. The default value is **FALSE**.

ChartShowBox
If **TRUE**, numbering and tick marks will be forced to appear on the edges of the plot area. For better visual reference, two extra axes will be drawn on both the top and right sides of the plot area. The default value is **FALSE.**

ChartTimeTrace
Setting this option 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. (See the paragraph on single variable monitoring). The default value is **FALSE**.

ChartStacked
If **TRUE**, 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. The default value is **FALSE**.

ChartAdjacent
If **TRUE**, all bars mapping to the same physical cell in the plot area will be made thinner, and shown side by side. This guarantees that any bar in a particular data cell can not hide a shorter bar in any lower numbered data set. For this option to work properly, all data sets should be set up with the same options. The default value is **FALSE**.

ChartXTicsInside, ChartYTicsInside
These options set the tick mark alignment with respect to the axis lines for X and Y axes. If **TRUE**, the bottom X-axis tick marks will be bottom justified, while top tics are top justified. Left Y-axis ticks will be left justified, while right ticks are right justified. The default values are **FALSE**.

ChartXTicsOutside, ChartYTicsOutSide
These options set the tick mark alignment with respect to the axis lines for X - and Y-axes. If **TRUE**, the bottom X-axis tick marks will be top justified, while top ticks are bottom justified. Left Y-axis ticks will be right justified, while right ticks are left justified. If neither or both of *Inside* and *outside* options are set, then tick marks are *centered* over the axis. The default values are **FALSE**.

ChartXGrid, ChartYGrid, ChartY2Grid
These options specify whether *grid lines* will be shown crossing the respective axis. The default values are **FALSE**.

ChartRescaleX, ChartRescaleY, ChartRescaleY2
These options specify as to whether an axis will be re-numbered (scaled) when one of the data points extends beyond its limit. Note that re-scaling will modify the tick mark, numbering, and grid line intervals to maintain a similar visual representation of the chart. If **`RescaleXOption`** is **FALSE**, data points falling beyond the limits of the X-axis will be discarded. A data point whose Y value is out of range of a non re-scaleable Y-axis is highlighted. The default value of these options is **TRUE**.

ChartCompressX
When this option is **TRUE**, 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. Setting the option to **FALSE** is equivalent to having data *scrolled* along the X-axis during rescaling. In this case, data scrolled out of view will be discarded. The default value is **FALSE**.

7.7.9 Setting Chart Data Set Options

Data set options apply only to a individual data set. Values in the **`DSOptionType`** enumeration specify the option to be changed. An option of a particular data set can be modified only AFTER that data set has been added to the chart.

```
ASK METHOD SetDSOption(IN dsNum : DSRefType);
(* possible option's shown below *)
IN option : ChartDSOptionType;
IN optionOn : BOOLEAN);
```

ChartDSDiscrete
If **TRUE**, the data set is represented by a fixed number of *data cells*. The width of a data cell is set with the **SetDSInterval** method. When a value is plotted to a discrete data set, the data bar whose X-center point is closest to the given X-coordinate will be modified. This type of data set can only accumulate as much data as there are cells in the chart. If this option is **FALSE**, the data set is *continuous* and will accumulate as much data as is plotted. In this case the value plotted will be the exact point specified to **SetCoordinate** or **Plot**. The default value is **TRUE**.

ChartDSStatic
Applies to continuous data sets only. This option should be enabled if the data to be plotted to the chart is known off-line (i.e. all data is to be plotted at the same time). This will dramatically improve plotting time for charts showing large amounts of static data. The default value is **FALSE**.

ChartDSSurface
Setting this option will show the data set as a continuous surface connecting successive data points. Surfacing can be enabled for both discrete and continuous data sets, (In the continuous case it should ALWAYS be enabled). The default value is **FALSE**.

ChartDSCentered
Applies to discrete data sets only. If **TRUE**, the center point of the bar is equal to the center point of the data cell. Generally, for histograms this option should be **TRUE**, and for surface and bar charts it should be **FALSE**. Default value is **TRUE**.

ChartDSNarrow
Applies to discrete (non-surface) data sets only. Makes bars thinner so that gaps can be seen in-between them. Should be **TRUE** for bar charts. The default value is **FALSE**.

ChartDSY2Axis
If **TRUE** this data set will belong to the second (right) Y-axis. In this case, Y-coordinate values passed to the **SetCoordinate** and **Plot** methods should fall within the Y2 axis boundaries. If this option is **FALSE** for all data sets, then the right Y-axis is not shown. The default value is **FALSE**.

ChartDSInterpolate
Applies to surface charts only. If **TRUE**, successive data points are connected by a straight line. Otherwise, a horizontal line will fill the gap. This line is positioned vertically at the leftmost data point. The default value of the option is **TRUE**.

ChartDSFill, ChartDSLine, ChartDSMark
These options determine which primitive types to use in showing the data set. Usually, only the *fill* option is enabled for histograms and bar charts. The default value for **`FillOption`** is **`TRUE`**, and for **`LineOption`** and **`MarkOption`** it is **`FALSE`**.

7.7.10 Setting Chart Fields

The chart *fields* are values of type **`REAL`** defining various intervals and boundaries pertaining to the axes. Numbering, tick mark and grid line intervals are expressed with respect to the range implied by the axis boundaries. The item is not be displayed if the interval for an item is zero. The **`SetAxisField`** method given below modifies a particular field in the chart. The fields available are described by the **`ChartFieldType`** enumeration.

```
ASK METHOD SetAxisField(
    IN field : ChartAxisFieldType;  (* possible fields
    described below *)
    IN fieldValue : REAL);
```

ChartXMin, ChartYMin, ChartY2Min, ChartXMax, ChartYMax, ChartY2Max
These are the boundaries of each axis. These values are displayed on the left and right of the X-axis, and the top and bottom of both Y-axes.

ChartXInterval, ChartYInterval, ChartY2Interval
These fields are the tick mark intervals. The X-tick interval is used for the discrete data set cell width if this value is zero.

ChartXMinorInterval, ChartYMinorInterval, ChartY2MinorInterval
These fields are for the smaller (minor) tick marks shown between the larger (major) ticks. Generally speaking, the major interval should be a multiple of the minor.

ChartXNumInterval, ChartYNumInterval, ChartY2NumInterval
These fields specify the interval between successive numberings on an axis.

ChartXGridInterval, ChartYGridInterval, ChartY2GridInterval
These fields specify the interval between successive grid lines on an axis. Grid lines are only shown if the appropriate option has been set to **`TRUE`**.

ChartXIntercept
Point along the X-axis where the Y-axis intersects.

ChartYIntercept
Point along the Y-axis where the X-axis intersects.

ChartX2Intercept
Point along the X-axis where the second (right) Y-axis intersects.

ChartXScale, ChartYScale, ChartY2Scale
These fields are the scale factors applied to the data. Coordinates passed to the **`SetCoordinate`** and **`Plot`** methods will be multiplied by these factors before being displayed on the chart. (However, the *Y()* method returns the original un-modified Y coordinate value). Default values are 1.0.

7.7.11 Chart Components Listed in the 'GraphPartType' Enumeration

ChartTitle, ChartXTitle, ChartYTitle, ChartY2Title
Specifies one of the text string labels around the plot area on the chart face. Text font, and color can be set.

ChartXAxis, ChartYAxis, ChartY2Axis
Refers to an axis including the axis line, and tick marks. Only the color attribute can be set.

ChartXNumbering, ChartYNumbering, ChartY2Numbering
Refer to all numbers on an axis. Text font and color can be set.

ChartXGrid, ChartYGrid, ChartY2Grid
Refers to the grid line for a particular axis. Line style and color attributes can be set.

ChartBorder
Refers to the rectangular 'background' encompassing the entire chart. Fill style and color attributes can be set.

ChartLegend
Refers to the text describing data set contents in all legends. Text font and color can be set.

ChartDataSet
Refers to the primitives composing the data set specified by the '**`dsnum`**' parameter. Line style, fill style, mark style and color can be set.

7.7.12 Other Methods of ChartObj

Other miscellaneous attributes of a chart can be set up using the following methods:

ASK METHOD SetAxisTitles(IN xAxisTitle, yAxisTitle : STRING);
ASK METHOD SetY2AxisTitle(IN y2AxisTitle : STRING);

These methods set the text "title" display along one of the three axes. Y -axis title text will be automatically rotated 90 degrees while the Y2 axis title will be rotated 270 degrees.

ASK METHOD AddDataSet;

Adds a new data set to the chart. Should be called once for every data set the chart is to contain. Must be called BEFORE a new data set can be referenced.

ASK METHOD RemoveDataSet(IN dsnum : DSRefType);

Removes a data set from the chart. All data contained in the data set is discarded and erased from the plot area. Higher numbered data sets will be consecutively renumbered.

ASK METHOD ClearDataSet(IN dsnum : DSRefType);

All data contained in the data set is discarded and erased from the plot area.

ASK METHOD SetDSTitle(IN dsnum : DSRefType; IN dataSetLegend : STRING);

Sets the text shown in the legend for the specified data set.

ASK METHOD SetDSInterval(IN dsnum : DSRefType; IN interval : REAL);

Sets the width of each data cell for a *discrete* data set between the successive data cells of a *discrete* data set.

ASK METHOD ClearDataSet(IN dsnum : DSRefType);

Discards all previously plotted data in the given data set.

ASK METHOD SetDSInterval(IN dsnum : DSRefType; IN width : REAL);

Sets the cell width along the X axis for a "Discrete" type data set. If this parameter is 0, then the cell width will be the X tick mark interval.

ASK METHOD SetDSTitle(IN dsnum : DSRefType; IN title : STRING);

Sets the data set description text shown in the legend.

7.7.13 Example of Program Code for Creating a Chart

Suppose we wanted to create a chart showing plots of the mathematical functions $y = t^2$ and $y = t^3$. The following code will accomplish this:

```
NEW(chart);
ASK chart TO SetTitle("Example of two continuous data
sets");
```

```
ASK chart TO SetAxisTitles("Time in seconds", "f(t)");
   { create two data sets }
ASK chart TO AddDataSet;
ASK chart TO AddDataSet;

  { set up options for chart and data set }
ASK chart TO SetOption(ChartShowLegend, TRUE);
ASK chart TO SetOption(ChartShowBox, TRUE);
ASK chart TO SetDSTitle(1, "Plot of y = t*t");
ASK chart TO SetDSTitle(2, "Plot of y = t*t*t");

  { set data set options }
FOR i := 1 TO 2
  ASK chart TO SetDSOption(ChartDSFill, i, FALSE);
  ASK chart TO SetDSOption(ChartDSLine, i, TRUE);
  ASK chart TO SetDSOption(ChartDSDiscrete, i, FALSE);
  ASK chart TO SetDSOption(ChartDSSurface, i, TRUE);
  ASK chart TO SetDSOption(ChartDSStatic, i, TRUE);
END FOR;

  { set chart fields }
ASK chart TO SetAxisField(ChartXMin, -1.0);
ASK chart TO SetAxisField(ChartYMin, -1.0);
ASK chart TO SetAxisField(ChartXMax, 1.0);
ASK chart TO SetAxisField(ChartYMax, 1.0);

ASK chart TO SetAxisField(ChartXInterval, 0.1);
ASK chart TO SetAxisField(ChartYInterval, 0.1);
ASK chart TO SetAxisField(ChartXNumInterval, 0.5);
ASK chart TO SetAxisField(ChartYNumInterval, 0.5);

  { plot values to the chart }
FOR i := -100 TO 100
  t := FLOAT(i) * 0.01;
  ASK chart TO SetCoordinate(1, t, t * t);
  ASK chart TO SetCoordinate(2, t, t * t * t);
END FOR;

  { finally display the chart }
ASK window TO AddGraphic(chart);
ASK chart TO SetScaling(0.5, 0.5);
ASK chart TO DisplayAt(8192.0, 8192.0);

  { wait for user to click on something }
```

Chapter 8: Controls

The MODSIM III Graphical User Interface (GUI) provides a connection to the underlying vendor toolkit. Each control is associated with an object and is used to accept user input. Different control types are provided to allow various types of input to be accepted.

CONTROL TYPES:

ButtonObj—Receives simple selection.

CheckBoxObj—Receives YES/NO input.

ComboBoxObj—A combo box that allows you to either select from a list or type in your own selection.

DialogBoxObj—Modal or modeless container of controls.

FileDialogBoxObj—Allows you to browse the file system and select a file.

FontDialogBoxObj—Allows you to browse and select installed text fonts.

LabelObj—Displays a **STRING** or a group box.

ListBoxItemObj—Label inside a list box which can be selected.

ListBoxObj—Allows selection of exactly one item out of many. Contains list box items.

ListBoxMultObj—Allows selection of a variable number of items. Contains list box items.

MessageDialogBoxObj—Receives simple yes, no input, or alerts users.

MenuBarObj—Contains menus.

MenuItemObj—Receives simple selection.

MenuObj—Contains menu items.

MultiLineBoxObj—Receives multi-line text input.

PaletteObj—Allows modeless selection from an array of one or two state buttons.

PaletteButtonObj—Receives simple selection or YES/NO input from inside a palette.

PaletteSeparatorObj—Separates groups of palette buttons.

popMenuObj—Contains menu items.

RadioBoxObj—Contains a group of radio buttons.

RadioButtonObj—Receives YES/NO input which is mutually exclusive among a group of radio buttons.

TableObj—Receives row/column input.

TextBoxObj—Receives text string input.

TreeObj—Allows selection of one item out of many.

TreeItemObj—Label inside a **TreeObj**.

ValueBoxObj—Receives numeric input.

Several objects are used as intermediate object types. They should not be used directly. These intermediate types include:

ControlVObj—Basic control object which all specific controls are derived from.

FormVObj—Object which is a parent to other controls but does not have a parent control. The **MenuBarObj** and **DialogBoxObj** are derived from this object.

ToggleVObj—Object used to receive ON/OFF input. **CheckBoxObj** and **RadioButtonObj** are derived from this object.

The control inheritance tree is as shown in figure 8-1.

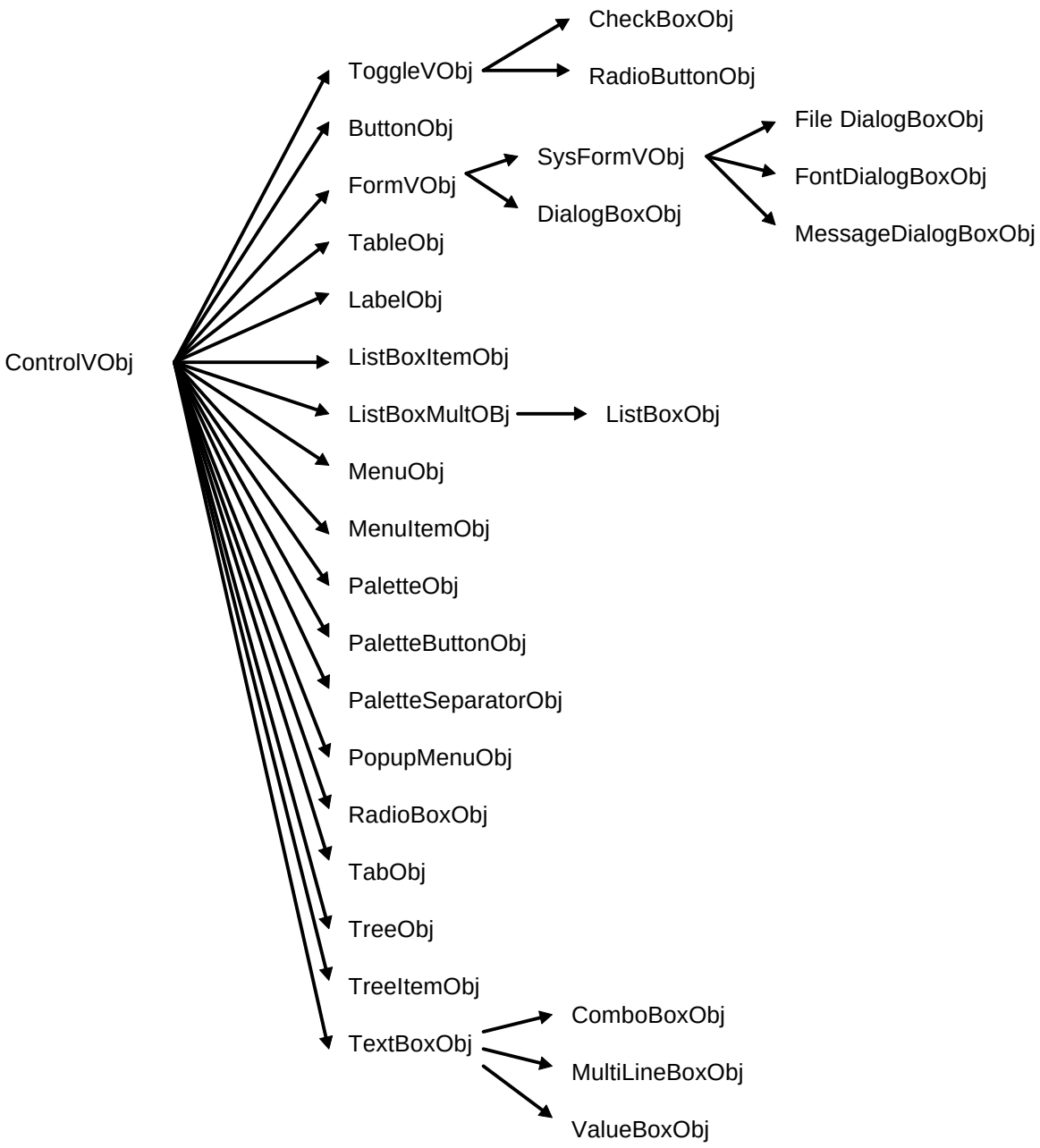


Figure 8-1. Inheritance Tree for Controls

8.1 Creating Controls

Controls are created interactively using the SIMGRAPHICS II editor. SIMDRAW stores a description of the controls it has created in a library file. Part of the control description includes a reference string and ID value which form a unique concatenated "key" used to identify the control within a MODSIM III program. Using the controls created by SIMDRAW involves loading the library description and finding the individual controls

using the reference string and ID. A control can be found using the following method:

```
control := ASK dialogbox Child(ReferenceName, id);
```

8.2 Retrieving Synchronous Input from Controls

After a **DialogBoxObj** has been asked to **AcceptInput,** the user may click on check boxes to change their state, enter text into text boxes, enter values into value boxes and press on buttons—in any order. At some point you will want to find out what the user has selected. **AcceptInput()** does not return until the dialog box is erased, which usually happens because a terminating button is pressed. **AcceptInput()** returns the contents of the **LastPicked** field after the selection has been made.

If you ask a menu bar to **AcceptInput()** the program will stop execution until one of the menu items has been selected:

```
item:= ASK MenuBar to AcceptInput()
CASE ASK item ReferenceName
   WHEN "file":
CASE item.LastPicked.ReferenceName
     When "exit": HALT;
   ...
END CASE;
```

To examine the state of a control, you interrogate a field of the object which contains its state information. Please note that the **TextBox** and **ValueBox BeSelected** methods are not called (unless their **ReturnSelectable** field is TRUE). You must use the **Text()** method of the text box to return its contents. **Value()** must be used to return the contents of a **ValueBox**.

8.3 Receiving Asynchronous Input from Controls

If your program needs immediate notification of control interaction (without waiting for **AcceptInput()** to return after a terminating button has been pressed) you can OVERRIDE the **BeSelected** method of the **DialogBoxObj, PaletteObj,** or **MenuBarObj**. The **BeSelected** method is called automatically when one of the following events happens:

1. The user clicks on a button, palette button, check box, radio button, list box item, tree item, or menu item.

2. The user presses **Return** inside a text box, combo box, or value box whose **ReturnSelectable** field is **TRUE**.

3. The user clicks on a cell inside a table.

Within the **BeSelected** method you can check the **LastPicked** field to see which control was just clicked on. To do this you'll need to create your own variety of **DialogBoxObj, MenuBarObj** or **PaletteObj**. For example:

```
MyDialogBoxObj = OBJECT(DialogBoxObj)
  OVERRIDE
    ASK METHOD BeSelected;
END OBJECT;
```

The implementation of **BeSelected** could look something like this:

```
OBJECT MyDialogBoxObj;
  ASK METHOD BeSelected;
  BEGIN
       { "browse", "itemlist" and "radiobox" are names
          assigned within SIMDRAW }
    CASE LastPicked.ReferenceName
      WHEN "browse" :
        OUTPUT("Browse button selected");
      WHEN "itemlist" :
        OUTPUT("List box item selected");
      WHEN "radiobox" :
        OUTPUT("Radio button selected");
      ...
    END CASE;
  END METHOD;
END OBJECT;
```

8.4 Drawing and Erasing

Any control may be drawn or erased using the **GraphicVObj Draw** and **Erase** methods. If the **FormVObj** containing the control has not been drawn then drawing and erasing a child control has no effect.

8.5 Deactivating and Activating

All controls can be activated or deactivated. If a control is deactivated it will not accept user input and is dimmed. A control is deactivated by the following **set/draw** methods:

```
ASK control TO SetSelectable(FALSE);
ASK control TO Draw;
```

or by the shortcut:

```
ASK control TO Deactivate;
```

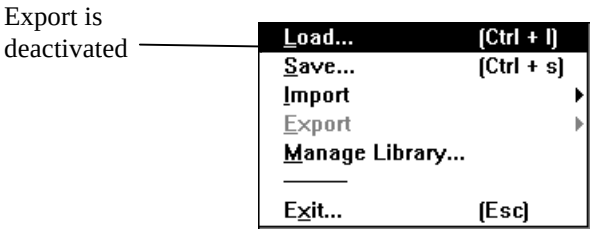


Figure 8-2. Deactivated Control

The control can be reactivated using the **Set**/**Draw** methods:

```
ASK control TO SetSelectable(TRUE);
ASK control TO Draw;
```

or by the shortcut:

```
ASK control TO Activate;
```

Activating a control which is already activated, or deactivating a previously deactivated control has no effect.

8.6 Setting the Control's Label

A control can have its optional label set using the **Set/Draw** methods:

```
ASK control TO SetLabel("label");
ASK control TO Draw;
```

8.7 Disposing Controls

Any control can be disposed using the **DISPOSE** procedure. Disposing a control automatically disposes of all the controls attached to it. In addition, disposing a window or a dialog box causes all of the controls attached to it to be disposed.

8.8 Updating Controls

Any control can be asked to draw. **Draw** causes the attributes of the control and any control attached to it to take effect.

8.9 Buttons

Figure 8-3. Button

A button is used to receive a simple input event. Clicking on the button selects it. It is also used to verify controls, and terminate dialog boxes.

Buttons can be verifying and/or terminating. A verifying button causes all value boxes in the dialog box to check their contents when it is pressed.

Buttons can also be terminating. A terminating button causes its parent dialog box to be erased when it is selected.

8.10 Check Box

Figure 8-4. Check Box

The check box is used to receive YES/NO input. The **Checked** field shows the state of the check box. You can set the state of the check box using the **SetCheck** method as follows:

```
ASK checkbox TO SetCheck(TRUE);
ASK checkbox TO Draw;
```

or by the shortcut:

```
ASK checkbox TO DisplayCheck(TRUE);
```

8.11 Text Box

Figure 8-5. Text Box

The text box is used to receive a string of characters.

The contents of the text box can be read using the following method:

```
textstring := ASK textbox Text();
```

The contents of a text box can be changed using the **Set**/**Draw** methods.

```
ASK textbox TO SetText("new contents");
ASK textbox TO Draw;
```

or by using the shortcut:

```
ASK textbox TO DisplayText("new contents");
```

8.12 Value Box

Figure 8-6. Value Box

A value box receives numeric data between a specified minimum and maximum. The data can be entered in standard or scientific notation. If the entered value does not fall within the valid range or is non-numeric, and the verify attribute has been turned on, the window will beep and a **<** sign will be placed next to the invalid value.

The contents of the value box can be read as follows:

```
value := ASK valuebox Value();
```

Value() returns a REAL number.

The content of a value box can be set using the **Set**/**Draw** methods:

```
ASK valuebox TO SetValue(1.222);
ASK valuebox TO Draw;
```

or by using the shortcut:

```
ASK valuebox TO DisplayValue(1.222);
```

8.13 List Box and List Box Item

Figure 8-7. List Box

The list box allows list box items to be selected. Any number of list box items may be attached to the list box control. There are two types of list boxes: a **ListBoxMultObj** allows any number of **ListBoxItemObjs** to be selected; a **ListBoxObj** allows exactly one **ListBoxItemObj** to be selected.

List box items are added and removed from the list box using the standard **GraphicSetObj, AddGraphic** and **RemoveGraphic** methods. The order of the list box items in the list box is determined by their order in the graphical set. The following code creates, initializes, and adds several items to a list box:

```
NEW(ListBoxItem1);
NEW(ListBoxItem2);
NEW(ListBoxItem3);
NEW(listbox);

ASK ListBoxItem1 TO SetLabel("item number 1");
ASK ListBoxItem2 TO SetLabel("item number 2");
ASK ListBoxItem3 TO SetLabel("item number 3");

ASK listbox TO AddGraphic(ListBoxItem1);
ASK listbox TO AddGraphic(ListBoxItem2);
ASK listbox TO AddGraphic(ListBoxItem3);
ASK listbox TO Draw;
```

Items can be removed using the standard **RemoveGraphic** or **RemoveThisGraphic** methods as follows:

```
ASK listbox TO RemoveThisGraphic(ListBoxItem1);
ASK listbox TO Draw;
```

If the items are added using the **AddAlpha()** method, then the items in the list box will appear in sorted order:

```
ASK listbox TO AddAlpha(ListBoxItem);
```

All items in a list box can be removed by doing the following:

```
ASK listbox TO RemoveAll;
```

The selection status of an item in a **ListBoxMultObj** can be set using either the listbox's or the listbox item's **SetIsSelected()** method:

```
ASK listbox TO SetIsSelected(ListBoxItem1, TRUE);
ASK ListBoxItem2 TO SetIsSelected(FALSE);
ASK listbox TO Update;
```

You can find out which items in a list box are currently selected by iterating through the children of the list box and examining their **IsSelected** fields:

```
VAR anItem : ListBoxItemObj;
...
anItem := ASK listbox FirstGraphic();
WHILE (anItem <> NILOBJ)
     IF (ASK anItem IsSelected) ....
     END IF;
     anItem := ASK listbox NextGraphic(anItem);
END WHILE;
```

The current list box selection in a **ListBoxObj** can be set as follows:

```
ASK listbox TO SetSelectedItem(ListBoxItem1);
ASK listbox TO Draw;
```

or by the short cut:

```
ASK listbox TO DisplaySelectedItem(ListBoxItem1);
```

The current selection in a **ListBoxObj** can be retrieved as follows:

```
currentitem := ASK listbox SelectedItem;
```

Detection of double clicks on list box items is usually used as a short cut for selecting an item, and then picking an edit button. **ListBoxObj** has the following field:

```
SecondClick : BOOLEAN;  { TRUE iff last item selected
                          was double clicked on }
```

To use this, the **ListBoxObj** or its parent **DialogBoxObj** should be subclassed and have its **BeSelected** method overridden. When the **BeSelected** method is called, the listbox's **SecondClick** field will be TRUE if the last item selected was double-clicked.

8.14 Radio Box and Radio Button

Figure 8-8. Radio Box

The radio box is a container for radio buttons. The position of the radio buttons are determined by the position of the radio box. Each of the radio buttons appears in a vertical column starting at the radio box position.

The currently selected radio button can be retrieved from either the radio box or from the individual radio buttons. The following retrieves it from the radio box:

```
currentbutton := ASK radiobox SelectedButton;
```

The selected button in a radio box can be set using the **Set/Draw** method:

```
ASK radiobox TO SetSelectedButton(button);

ASK radiobox TO Draw;
```

or by the shortcut

```
ASK radiobox TO DisplaySelectedButton(button);
```

8.15 Tree View Control

Tree view control (figure 8-9) is a special list box control that displays a set of objects as an indented outline based on their logical hierarchical relationship. The control includes buttons that allow the outline to be expanded and collapsed. You can use a tree view control to display the relationship between a set of containers or other hierarchical elements. (See the Windows95 Explorer for an example).

The tree view control itself is implemented by creating a **TreeObj** (found in **Gtree**). The **TreeObj** contains **TreeItemObj** objects representing each item in the list. Each item is shown with a text label and a bitmap icon. The hierarchy is built by adding **TreeItemObj**s to other **TreeItemObj** objects. A **TreeObj** is defined below:

```
SelectedItem  : TreeItemObj
  { Currently selected list box item. };
Width : INTEGER  { Width of the list box in font units. };
```

```
Height : INTEGER  { Height of the list box in font
                    units. };
Maximized : BOOLEAN  { Tree will be maximized upon
                       creation };
SecondClick : BOOLEAN   { Was last TreeItemObj
                          selection a double click? };
```

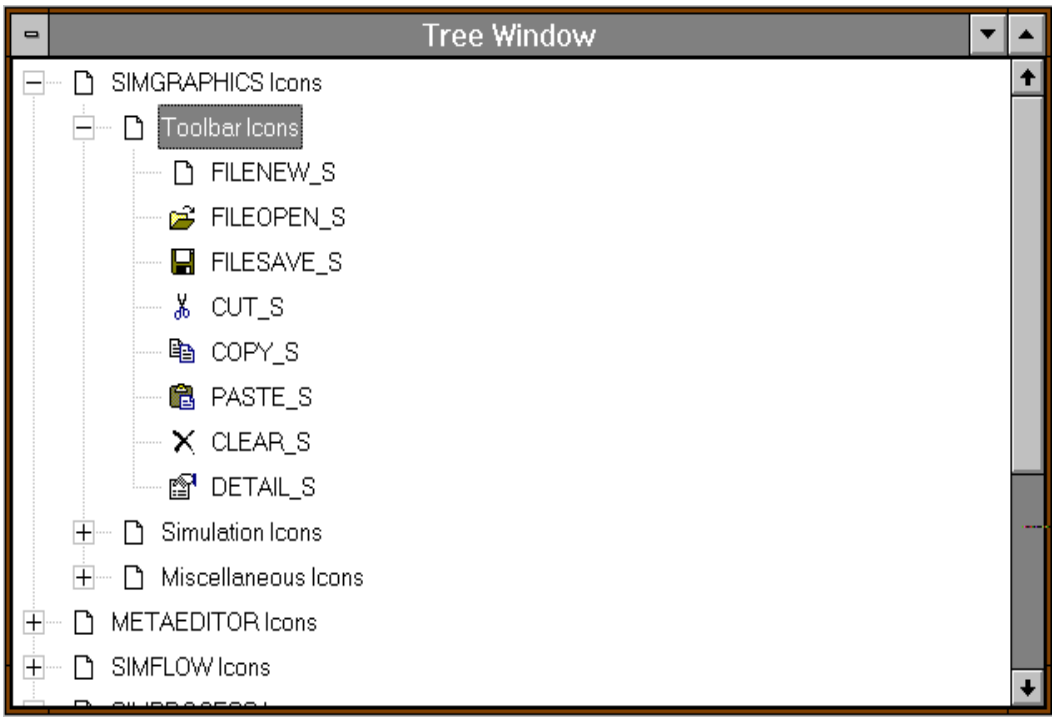


Figure 8-9. Tree View Control

ASK METHOD SetIsSelected(IN item : TreeItemObj;
IN isSelected : BOOLEAN);
SetIsSelected of item. Assumes item is in the graphical set.

ASK METHOD DisplayIsSelected(IN item : TreeItemObj;
IN isSelected : BOOLEAN);
Selects an item in the tree and updates the tree display. Assumes item is in the graphical set belonging to the **TreeObj** and the **TreeObj** has already been displayed.

ASK METHOD SetSize(IN width, height : INTEGER);
Sets the width and height of the tree in font units.

ASK METHOD SetMaximized(IN maximized : BOOLEAN);
Sets the tree to become maximized within a **ControlWindowObj**. This method only has an effect when the tree is attached to a **ControlWindowObj**. This method invalidates any calls to **Set/Size**.

A **TreeItemObj** defines the following fields and methods:

```
IsSelected  : BOOLEAN { TRUE if this item is currently
   selected. };
IconName    : STRING  { Icon name to appear in the
   TreeObj adjacent to this TreeItemObj. };
```

ASK METHOD SetIconName(IN iconname : STRING)
Sets the resource or file name of the bitmap to be used in the face of the button. In MS Windows systems, this refers to the name of the resource containing the bitmap data. On X Window systems, it is the name of the window dump ("`.xwd`") file containing the bitmap. A path can be prepended to this file name, but the extention should NOT be included.

ASK METHOD SetIsSelected(IN isselected : BOOLEAN);
Notify the parent tree and show this item as either selected or not-selected.

ASK METHOD SetExpanded(IN expand : BOOLEAN);
Expands or contracts the list of associated child items. There is no effect if the tree item has no children.

8.16 LabelObj

A **LabelObj** displays a non-user editable **STRING** in a dialogbox. You can set the text of a **LabelObj** using the **SetLabel()** method. The position of a label can be set with the **SetTranslation()** method; the **Translation** corresponds to the location of the upper left corner of the label.

8.17 ComboBoxObj

SIMGRAPHICS II supports a combination box control (figure 8-10). A combo box resides in a dialog box, and is composed of an editable text field and a drop down list. When the button to the right of the text field is clicked on, the list of choices for that field appears allowing the user to make a selection. The selected list item is then displayed in the text field.

The list of options can be defined programmatically or from within SIMDRAW. This list can be sorted alphabetically and the text field is optionally editable. A **ComboBoxObj** is derived from a **TextBoxObj** and is described below.

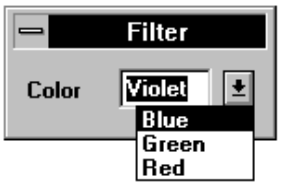


Figure 8-10. Combo Box

ASK METHOD SetOptions(IN options : OptionListType);
Sets the items in the drop down option list to the text values contained by the given array of **STRINGs.**

ASK METHOD SetHeight(IN height : INTEGER);
Sets the number of option items that can be concurrently viewed in the drop down list. If the number of items exceeds **"height"** then the list will be scrollable.

ASK METHOD SetEditable(IN editable : BOOLEAN);
Sets whether or not the text shown in the text field can be edited by the user. If **editable** is **FALSE**, then this field can only be set by selecting one of the displayed options**.**

ASK METHOD SetSorted(IN sorted : BOOLEAN);
Sorted sets whether of not the items in the option list will appear alphabetically.

The following **TextBoxObj** methods are useful for a **ComboBoxObj**:

ASK METHOD SetWidth(IN width : INTEGER);
ASK METHOD Text() : STRING;
ASK METHOD SetText(IN text : STRING);
ASK METHOD SetReturnSelectable(IN selectWithEnter : BOOLEAN);
The **BeSelected** method is called whenever an item in the option list is selected. The currently selected/displayed item can be retrieved using the **Text()** method.

Note: If the **ReturnSelectable** flag is **TRUE**, then **BeSelected** will be called whenever the user presses the **Return** key while the text field has input focus.

8.18 MultiLineBoxObj

A fully editable multi-line text box can be created and added to your dialog box. It can contain any number of lines of readable / writable text. Text box contents can be set by passing an array of**STRING**s to the **SetTextBuffer()** method. For example:

Figure 8-11. Multi-line Text Box

```
FROM GTypes  IMPORT TextBufferType;
...
VAR
  buffer : TextBufferType;
  multiLineBox : MultiLineBoxObj;
...
NEW(buffer, 1..3);
buffer[1] := “The quick brown foxes”;
buffer[2] := “jumped over”;
buffer[3] := “lazy dog’s back.”;
ASK multiLineBox TO SetTextBuffer(buffer);
ASK multiLineBox TO Draw;
```

You can get the current contents of the box with the **`TextBuffer()`** method. Do not DISPOSE of the array returned by this method. Use the **`CLONE`** operation if the contents are to be saved after the multi-line box has been destroyed. For example:

```
...
buffer := CLONE(ASK multiLineBox TextBuffer());
FOR I := LOW(buffer) TO HIGH(buffer)
  OUTPUT(i, “ : “, buffer[i]);
END FOR;
```

`ASK METHOD SetSize(IN numColumns, numRows : INTEGER)`

Sets the width and height (in character units) of the text box. This includes the size of the scroll bars

`ASK METHOD SetTextBuffer(IN lines : TextBufferType)`

Sets the entire contents of the Multi-line box. Each element in the given array corresponds to a line of text.

`ASK METHOD TextBuffer() : TextBufferType`

Returns the text buffercontaining all current text.

CAUTION: The buffer will be DISPOSEd of when this object is disposed. Use **`CLONE`** if this data is to be saved.

ASK METHOD SetHorizontalScrolling(IN HorzScrolling : BOOLEAN)
Selects between horizontal scrolling or automatic line wrapping. If **TRUE**, a horizontal scroll bar will be added to allow viewing long lines. **FALSE** will cause long lines to automatically wrap at word boundries.

ASK METHOD SetMaximized(IN maximized : BOOLEAN)
Sets the control to become maximized within a **ControlWindowObj** when it is created. This method only has an effect when the parent window is a **ControlWindowObj**. This method invalidates any calls to **SetSize**.

Note: The **BeSelected()** method is *not* called when the user presses the **Return** key. However, the **BeModified()** method will be called whenever any change is made to the text contents by the user. The multi-line text box can be maximized to create a *text edit window*, if it has been added to a **ControlWindowObj** object. This is performed via the **SetMaximized()** method.

8.19 TableObj

A “Table” is a two dimensional array of selectable cells or text labels. The number of columns and rows of cells can be defined both programmatically and through SIMDRAW. Also definable is the width of each column and the size of the visible portion. If the table size exceeds its visible size in width or height, the table is automatically made scrollable in that direction. The left most column is numbered "1" while the top most row is numbered "1".

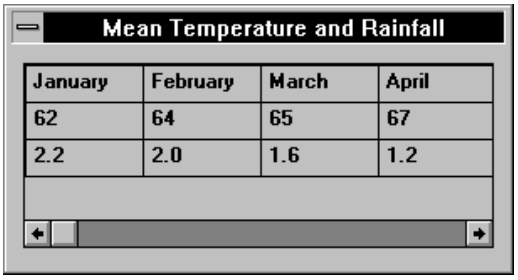


Figure 8-12 Table

A table can also have row and/or column headers. Column headers are shown by a row of cells on top of the table. Column headers scroll in the horizontal direction but remain fixed vertically (referred to as row 0). The row headers are a column of cells attached to the left side of the table which scrolls in the vertical direction only (referred to as column 0). Cells for the column and row headers can be selected and defined just like cells in the table content area.

The user can use the keyboard arrow keys (left, right, up, down) to navigate through a table. Each time focus is changed to a different cell using the arrow keys, the method will be called. If the **VerticalGrow** option is turned on and the DOWN arrow key is pressed when the focus is on the last row in the table, a new row will automatically be

added to the table. At this time the **BeExpanded** method is called to inform the program that the table has increased in height. If a non-arrow key is pressed when the focus is on a table cell, the **Keypress** method is called. This method can then be used to change focus to a text box allowing you to set the text of the selected cell without the user having to use the mouse. The following code can be used to aid in implementing a mechanism for allowing a user to easily set the contents of a table:

```
{ OVERRIDE the Keypress() method on a TableObj }
ASK METHOD Keypress(IN in keyPressed : STRING);
BEGIN
  ASK textbox TO DisplayText(keyPressed);
  ASK textbox TO ReceiveFocus;
END METHOD;

  { OVERRIDE the BeSelected() method on the TableObj }
ASK METHOD BeSelected;
BEGIN
  INHERITED BeSelected;
   ASK textbox TO DisplayText(Text(SelectedColumn,
   SelectedRow));
END METHOD;

  { OVERRIDE the BeSelected method on a TextBoxObj
   which accepts the text }
ASK METHOD BeSelected;
BEGIN
  INHERITED BeSelected;
   ASK table TO SetText(Text(), table.SelectedColumn,
   table.SelectedRow);
END METHOD;
```

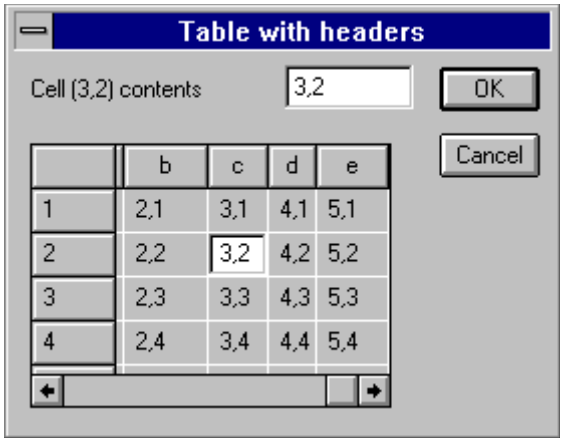


Figure 8-13. Table with Headers

TableObj defines the following **ASK METHOD**s:

SetText(IN text : STRING; IN column, row : INTEGER);
Sets the text shown in the cell at the given column and row.

Text(IN row, column : INTEGER) : STRING;
Returns the text in the cell at the given column and row.

SetSize(IN numColumns, numRows : INTEGER);
Sets the number of columns and rows of cells in the table.

SetVisibleSize(IN width, height : INTEGER);
Sets the size (in font units) of the visible portion of the table (not including scrollbars).

SetColumnWidth(IN columnNumber, width : INTEGER);
Sets the width (in font units) of the given column number.

ASK METHOD SetColumnAlignment(
IN columnNumber : INTEGER;
IN align : TextHorizType)
Sets the text alignment of all cells in the specified column. Only call this method after **SetSize**.

ASK METHOD SetHeadings(
IN haveRowHeadings, haveColumnHeadings : BOOLEAN)
Sets whether special scrolling panes containing row or column headings will be included with the table. The default is 'no headings'.

ASK METHOD SetVerticalGrow(IN addRowsWithScroll : BOOLEAN)
Sets whether rows will be automatically added to the bottom of the table when the user attempts to scroll vertically past the last row. The default is 'no headings' .

ASK METHOD Keypress(IN keyPressed : STRING)
Callback for when a key is pressed. Called automatically whenever a character is typed while focus is set to one of the table cells. A string representing the activated key is provided. (Generally, this string will be of length '1'.)

ASK METHOD BeExpanded
Called automatically whenever an expandable table is increased in size through the use of the down arrow key.

ASK METHOD SetMaximized(IN maximized : BOOLEAN)
Sets the table to become maximized within a **ControlWindowObj** when it is created. This method only has an effect when the parent window is a **ControlWindowObj**. This method invalidates any calls to **SetSize**.

The **BeSelected** method is called whenever a cell is clicked on. The **SelectedColumn** and **SelectedRow** fields can then be used to take the appropriate action.

8.20 Calling BeSelected for the TextBoxObj, ValueBoxObj and ComboBoxObj

The **BeSelected** method of a **TextBoxObj**, **ValueBoxObj**, and **ComboBoxObj** will be called whenever the **Return** key is pressed while typing into a field, if the **ReturnSelectable** field is set to true. If this behavior is desired, call the **SetReturnSelectable** method passing **TRUE,** or set the **ReturnSelectable** checkbox from within SIMDRAW

8.21 Dialog Box

The dialog box is a container for buttons, check boxes, list boxes, radio boxes, and labels. The size of the dialog box automatically adjusts to just fit its contents. The dialog box appears centered in its parent window by default.

The dialog box is attached to a window by:

```
ASK window TO AddGraphic(dialogbox);
```

After a dialog box is attached to a window and drawn it can receive input from the user. This is done using the **AcceptInput** method:

```
button := ASK dialogbox TO AcceptInput();
```

After a terminating button is selected the dialog box is erased and the terminating button is returned.

To receive dialog box selections while a simulation is running perform the following steps:

1. Create your own dialog box object and override its **BeSelected** method:

   NEW(MyDialogBox);

2. Ask the dialog box object to load its graphic description from a library:

   ASK MyDialogBox TO LoadFromLibrary(lib, "DialogBox");

3. Attach the dialog box to the window:

   ASK window TO AddGraphic(MyDialogBox);

4. Start the simulation:

   StartSimulation.

5. The **BeSelected** method of **MyDialogBox** will be called whenever there is a control selection, and the **LastPicked** field of the dialog box will point to the control which was selected. The **BeSelected** method should be overridden to look something like this.

```
ASK METHOD BeSelected;
VAR
   control : ControlVObj;
BEGIN
  control := LastPicked;
  CASE ASK control ReferenceName
    WHEN "MyCheckbox" :
    ...
    WHEN "MyRadioBox" :
    ...
    OTHERWISE
   END CASE;
END METHOD;
```

Dialog boxes are positionable independently of their parent window's position. It is not necessary to use dummy windows to position dialog boxes at various screen positions. **DialogBoxObj** has the following method:

```
ASK METHOD SetPositioning (IN align : DBPositionType) ;
{   Sets the Positioning field.
    Centered     -> Dialog box is centered within its
                    parent window.
    Other values -> Dialog box is positioned relative
                    to its Translation.
    BottomLeft positioning means the bottom left corner
will be located at the position of the screen specified by
the Translation.  Note that the Translation is specified
in screen coordinates, as is done for WindowObj's; that
is, (25.0, 75.0) would refer to the point on the display
1/4 of the screen width right of and 3/4 of the screen
height up of the bottom left corner of the screen.
    Positioning default value = Centered. }
```

Use methods **SetTranslation** and **SetPositioning** to specify a reference point on the screen and an alignment relative to that point, respectively. Note that the units used in **SetTranslation** are screen units, as used when positioning **WindowObj**s.

8.21.1 User-controlled Dialog Box Fonts

Microsoft Windows users can specify the font used for dialog box labels and controls, using a command line argument of the form:

```
-font <font family> [<size> [bold]]
```

where `<font family>` is a string (quoted with double quotes if it contains embedded spaces), `<size>` is an optional integer point size and **`bold`** is optional. It is currently not possible to specify bold without specifying the point size also. The following are all valid examples:

```
-font "MS Sans Serif"
-font "MS Sans Serif" 14
-font "MS Sans Serif" 12 bold
```

8.21.2 Tabbed Dialogs

An application can define multiple logical *pages* or sections of information within a single dialog by using fonts. This will help reduce the *clutter* in a dialog, and also cut down on the multiple levels of dialog nesting otherwise needed. Pages of controls in a **Tabbed Dialog** can be brought to the front by selecting its *tab* at the top of the page.

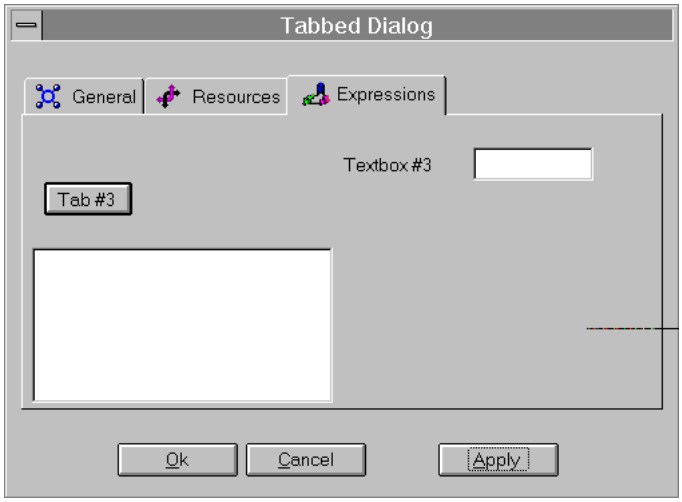


Figure 8-14. Tabbed Dialog

Tabbed Dialogs can be created in SIMDRAW or programatically. In SIMGRAPHICS II a tab page is implemented through a **`TabObj`** object found in the module **`GTab`**. **`TabObj`** objects are added directly to the **`DialogBoxObj`** while controls are added to the appropriate **`TabObj`**. The tab area of the dialog box is always positioned at the top left hand corner and cannot be moved. However, the tab area's *size* can be changed using the following method of the **`DialogBoxObj`**:

ASK METHOD SetTabSize(IN width : REAL; IN height : REAL);
Sets the height and width in font units of the tab area.

Fields and methods of the **TabObj** (found in the module **Gtab**) are defined as follows:

IconName : STRING;
Name of the bitmap shown in tab part of the page.

ASK METHOD SetIconName (IN Name : STRING);
Sets the name of the icon that appears to the left of the text in the tab. In MS Windows systems, this refers to the name of the resource containing the bitmap data. On X Window systems, it is the name of the window dump (".xwd") file containing the bitmap. A path can be prepended to this file name, but the extention should NOT be included. (Path names will be ignored on MS Windows).

The **SetLabel** method is used to set the text appearing in the tab.

8.21.3 Dialog Box Example Program

The next example accepts input from a dialog box created by SIMDRAW. The dialog box contains **OK** and **Cancel** buttons. It is displayed until one of the buttons is selected. **"Ok was selected"** or **"Cancel was selected"** is output.

```
MAIN MODULE Example7;
    { Simple dialog box. }

FROM Window  IMPORT WindowObj;
FROM Graphic IMPORT GraphicLibObj;
FROM Form    IMPORT DialogBoxObj;
FROM Button  IMPORT ButtonObj;
FROM Value   IMPORT ValueBoxObj;

VAR
   window : WindowObj;
   dialog : DialogBoxObj;
   button : ButtonObj;
   library : GraphicLibObj;
   valuebox : ValueBoxObj;

BEGIN
```

```
      { Initialize graphics and create a window. }
       NEW(window);
       ASK window TO Draw;

      { Load description of menu bar from the library. }
      NEW(library);
      ASK library TO ReadFromFile("Example7.sg2");
      NEW(dialog);
       ASK window TO AddGraphic(dialog);
       ASK dialog TO LoadFromLibrary(library,
          "DialogBox");
        valuebox := ASK dialog Child("capacity ", 0);
      ASK valuebox TO SetValue(47.3);
         { Wait for terminating button to be selected
           before exiting. }
      button := ASK dialog TO AcceptInput();

      CASE ASK button ReferenceName
        WHEN "ok"     :
         OUTPUT("Capacity  : ", valuebox.Value());
         OUTPUT("Ok was selected);
        WHEN "cancel" :
        OUTPUT("Cancel was selected");
        OTHERWISE
      END CASE;

      DISPOSE(window);

  END MODULE.
```

8.21.4 System File Browser Dialogs

The System File Browser dialog boxes take on the look and feel of the underlying graphical environment. It provides an easy method for saving and loading files, allowing the user to browse the file system. Below are the detailed descriptions of this object.

```
SysFormVObj = OBJECT(FormVObj)
  ASK METHOD AcceptSysInput() : BOOLEAN;
{   Accepts input on a system dialog box.  Control will not
be returned until the user selects the OK or CANCEL
buttons on the form.  TRUE is returned on selection of the
```

```
OK button, FALSE means that the CANCEL button was pressed.
}

FileDialogBoxObj = OBJECT(SysFormVObj)
  File : STRING  { Name of file selected during last
                   interaction. }
FilterList : NameListType     { List of filter groups };

ASK METHOD SetFilterList (IN filterList : NameListType)
  { Set a list of filter strings.  Supplies an optional
   list of filters and associated descriptive text.  The
   effect is system-dependant.   For Example:

    NEW(list, 1..3);
    list[1] := “Windows bitmap,*.bmp”;
    list[2] := “Windows metafile,*.wmf”;
    list[3] := “AutoCAD dxf,*.dxf”;

    ASK fileDialog TO SetFilterList(list);
  }

Path : STRING; { Path to selected file }
  Filter :STRING;{ File name selection mask for browse}

  ASK METHOD SetFilter(IN filterOrDefaultFile :STRING);
{   Sets the string that will be used as an initial file
name "filter" for browsing through the file system.  The
'*' can generally be used as a wild card, but some systems
may also support regular expressions. A default file name
can be given instead of the filter. }
```

Notice that a method, **AcceptSysInput**, is provided in module form for use with both **System Save** and **Load Dialog** boxes, and **Font Browser Dialog** boxes. Use this routine, instead of **AcceptInput**; it returns a boolean value indicating whether **OK** or **Cancel** was selected.

8.21.5 System Font Dialog

The standard system font selection dialog is available. On Motif and OPEN LOOK a generic font specification dialog is provided. The **FontDialogBoxObj** is included in the module Form:

```
SysFormVObj = OBJECT(FormVObj)
  ASK METHOD AcceptSysInput() : BOOLEAN;
```

```
  { Accepts input on a system dialog box.  Control will
   not be returned until the user selects the OK or CANCEL
   buttons on the form.  TRUE is returned on selection of
   the OK button, FALSE means that the CANCEL button was
   pressed.
  }

FontDialogBoxObj = OBJECT(SysFormVObj)
    {   This object represents a system-specific font
    browser dialog box.  Family, Size, Weight and Slant
    can be set using the SetInitialFont method before
    asking the dialog box to AcceptSysInput; the dialog
    box selection listboxes will be positioned to these
    values when it appears initially.  If Family is a nil
    string, the dialog box listboxes will come up in a
    default configuration.

    When interaction is completed, the Family, Weight,
    Size and Slant fields be used in the TextObj method
    SetSysFont. }

    Family : STRING ;  { System font family }
    Size   : INTEGER ; { System font size, in points }
    Weight : INTEGER ; { System font weight;
                         0 = lightest,
                         50 = normal; 100 = boldest }
    Slant  : INTEGER ; { System font slant; 0 = normal;
                         1 = italics }

  ASK METHOD SetInitialFont (IN family : STRING ;
          IN size : INTEGER ; IN weight : INTEGER ;
          IN slant : INTEGER ) ;
    {Use this method to set the initial font to be
    displayed in the browser when it is next drawn. }
```

Notice that the method **AcceptSysInput** is provided for use with both **System Save**, **Load Dialog**, and **Font Browser** dialog boxes. Use this routine, instead of **AcceptInput**; it returns a boolean indicating whether **OK** or **Cancel** was selected.

8.22. Alert Boxes

SIMGRAPHICS II supports interfaces to the standard built-in alert, information, and

question message boxes found under the MS Windows and X Windows toolkits. These are implemented through the **MessageDialogBoxObj.** The message box can be of the **OK** only, **OK/Cancel**, **Yes/No**, **Yes/No/Cancel**, **Retry/Cancel**, or **Abort/Retry/Ignore** variety. Any of the buttons on a message box can be the default button. The **AcceptSysInput()** method should be used to display the dialog and wait for a reply. This method will return **TRUE** if either the **Ok, Yes** or **Retry** buttons are clicked. A **Selected Button** field contains the enumeration value of the selected button after returning from **AcceptSysInput().** Message dialogs are modal, thereby requiring a response before mouse or keyboard interaction with any other objects can proceed. Figure 8-15 is an example of a **MessageDialogBoxObj**.

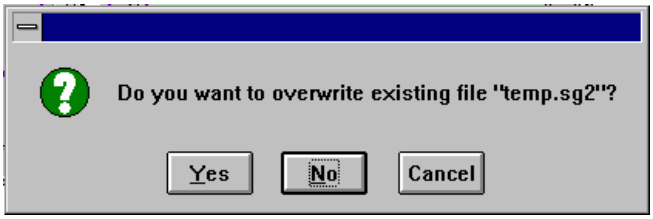


Figure 8-15. Alert Dialog Box

MessageStyleType = { use the "**SetStyle**" method to set which box to display } (PlainMessage, StopMessage, QuestionMessage, AlertMessage, InformationMessage);

MessageResponseType = { use the "**SetResponses**" method to set buttons to show } (OkResponse, OkCancelResponse, YesNoCancelResponse, YesNoResponse, RetryCancelResponse, AbortRetryIgnoreResponse);

MessageButtonType = { use the "**SetDefaultButton**" method to set default } (OkButton, CancelButton, YesButton, NoButton, AbortButton, RetryButton, IgnoreButton);

MessageDialogBoxObj defines the following **ASK METHOD**s:

ASK METHOD SetText (IN message : STRING)
Sets the message to be displayed in the dialog.

ASK METHOD SetTextBuffer (IN messageLines : TextBufferType)
Sets the message to be displayed in the dialog. The message is provided as an array of text. Each element is a line of text in the message.

ASK METHOD SetStyle (IN style : MessageStyleType)

Sets the nature of the message. An icon may be subsequently shown with the dialog alerting the user as to this nature. Appearance is system dependant.

ASK METHOD SetResponses (IN buttonSet : MessageResponseType);

Defines the set of response buttons on dialog.

ASK METHOD SetDefaultButton (IN defButton : MessageButtonType);

Defines which of the buttons is default. The default button can be activate d by pressing **Return.**

EXAMPLE:

The following code will show how to create and display a simple **Yes/No/Cancel** box, possibly responding to a user's request to exit the application.

```
VAR
   messageBox : MessageDialogBoxObj;

   NEW(messageBox);
   ASK messageBox TO SetText("Save  changes to file
   ""junker.dat"""?);
   ASK messageBox TO SetStyle(QuestionMessage);
   ASK messageBox TO SetResponses(YesNoCancelResponse);
   ASK messageBox TO SetDefaultButton(CancelButton);
   ASK window TO AddGraphic(messageBox);

   bool := ASK messageBox TO AcceptSysInput();
   CASE messageBox.SelectedButton
        WHEN YesButton : OUTPUT("Exit and save the
        changes!");
        WHEN NoButton :  OUTPUT("Exit, but dont save the
        changes!");
        WHEN CancelButton: OUTPUT("Forget it. Don't
        exit");
   END CASE;
```

8.23 Menu Bar, Menu, Menu Item

The menu bar (figure 8-16) is a container for menus. Menus are containers for menu options. Selecting a menu causes the menu options to appear. The menu bar is attached to a window by:

```
ASK window TO AddGraphic(menubar);
```

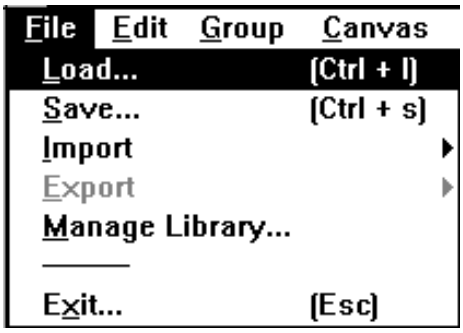


Figure 8-16. Menu Bar

After a menu bar is attached to a window and drawn it can receive input. This is done using the **AcceptInput** method:

```
item := ASK menubar TO AcceptInput();
```

AcceptInput returns the menu option which was picked. You may perform various actions depending on which item was selected.

```
CASE ASK item ReferenceName
   WHEN "Exit" : HALT;
   WHEN "Redo" : GoAgain;
   ...
   OTHERWISE
END CASE;
```

To receive a menu selection while a simulation is running you can do the following:

1. Create your own menu bar object and override the menu bar's **BeSelected** method:

   ```
   NEW(MyMenuBar);
   ```

2. Ask the menu bar object to load its graphic description from the library:

   ```
   ASK MyMenuBar TO LoadFromLibrary(lib, "MenuBar");
   ```

3. Attach the menu bar to the window:

   ```
   ASK window TO AddGraphic(MyMenuBar);
   ```

4. Start the simulation:

   ```
   StartSimulation.
   ```

5. The **BeSelected** method of **MyMenuBar** will be called whenever there is a menu selection, and the **LastPicked** field of the menu bar will point to the item which was selected. The **BeSelected** method should be overridden to look something like this:

```
ASK METHOD BeSelected;
VAR
 BEGIN
    CASE ASK LastPicked ReferenceName
     WHEN "Exit" : HALT;
     WHEN "Redo" : GoAgain;
     ...
   END CASE;
END METHOD;
```

8.23.1 Mnemonics

Mnemonics are one character keystrokes that enable you to pull down a menu, or select a menu option, using the keyboard while the menu or menu option is visible. On most systems, the mnemonic that activates a menu or menu option is displayed by underlining the mnemonic character in the menu string. On most systems, you would press **Alt** plus the mnemonic to pull down a menu, and then another mnemonic to select one of the items in the menu. Both **MenuObj** and **MenuItemObj** can be assigned mnemonics, either using SIMDRAW or using the SIMGRAPHIC's method, **SetMnemonic()**. Of course, no two **MenuItemObj**s in the same **MenuObj** should be assigned the same mnemonic. If your toolkit does not support mnemonics, then assigning them has no effect. The following code assigns the **File** menu the mnemonic "**f**", and the **Save** item in the **File** menu the mnemonic "**s**":

```
ASK FileMenu TO SetMnemonic("f");
ASK SaveItem TO SetMnemonic("s");
```

8.23.2 Check/Uncheck Menu options

A small check mark can be displayed next to the right of the label on a menu option. This is useful for indicating to the user whether an option in your program is currently "on" or "off". Check marks are displayed and hidden using the **SetCheck** and **DisplayCheck** methods. The following code puts a check mark next to a menu option label:

```
ASK menuItem TO DisplayCheck(TRUE);
```

8.23.3 Accelerators

MenuItemObj can be assigned an Accelerator, a set of keys that selects a menu choice even when the choice is not visible. Of course, no two **MenuItemObjs** in the same **MenuBarObj** should be assigned the same accelerator. Accelerators are usually displayed in a menu option by displaying a descriptive string to the right of the menu string. If your toolkit does not support accelerators, then assigning them has no effect.

The key press that makes up the accelerator can be any combination of the **Alt** key, the **Ctrl** key, and a printable ascii character or one of the function keys, written as **f1**, **f2,** etc. You must also specify the descriptive string that appears to the right of a menu string. The following code assigns an accelerator for item1 and item2; to activate item1, you would hold down the **Ctrl** key while typing **a**; to activate item2 you would press **f 2**.

```
ASK item1
 TO SetAccelerator(FALSE, TRUE, "a", "ctrl+a");
ASK item2
 TO SetAccelerator(FALSE, FALSE, "f2", "funkey 2");
```

8.23.4 Menu Bar Example Program

The following example accepts input using a menu bar created by SIMDRAW. When print is chosen **"Print was selected"** is output. Selecting exit terminates the program.

```
MAIN MODULE Example6;
 { Simple menu bar. }

FROM Window  IMPORT WindowObj;
FROM Graphic IMPORT GraphicLibObj;
FROM Menu    IMPORT MenuBarObj, MenuItemObj;

VAR
   window    : WindowObj;
   menubar   : MenuBarObj;
   item      : MenuItemObj;
   library   : GraphicLibObj;
   Done      : BOOLEAN;

BEGIN

      { Initialize graphics and create a window. }
    NEW(window);
    ASK window TO Draw;
```

```
    { Load description of menu bar from the library.}
  NEW(library);
  ASK library TO ReadFromFile("Example6.sg2");
  NEW(menubar);
  ASK menubar TO LoadFromLibrary(library, "MenuBar");

    { Add menubar to window and display it. }
  ASK window  TO AddGraphic(menubar);
  ASK menubar TO Draw;

    { Wait until exit is selected before exiting. }
    WHILE NOT Done
      item := ASK menubar TO AcceptInput();
      CASE ASK item ReferenceName
        WHEN "Exit"  : Done := TRUE;
        WHEN "Print" : OUTPUT("print selected");
        OTHERWISE
      END CASE;
    END WHILE;

      { Dispose window and all attached objects. }
    DISPOSE(window);

END MODULE.
```

8.23.5 Cascadable Menus

Menus in SIMGRAPHICS II can be arranged hierarchically with an unbounded depth. In other words, a **MenuObj** can contain menu options *and* other menus. A menu contained in another menu will usually be shown with a small arrow **(==>)** and can be brought down by dragging the mouse over this arrow and to the right. When a menu option contained by one of the menus is selected, the **BeSelected** methods for all container menus will be called in a bottom up fashion. Cascadable menus can be created programmatically with the **AddGraphic** and **AddChild** methods, or by using SIMDRAW.

8.23.6 Popup Menus

A single-click on an object in the canvas of a **WindowObj** can bring up a pop-up *context* menu containing commands that apply to the object. Examples for commands appearing on this menu would be **Print**, **Cut**, **Copy, Paste, Delete, Rename** and **Properties**. A popup menu is implemented through a **PopupMenuObj** object (found in the **Menu** module) and

contains **MenuItemObj** objects. A popup menu can be displayed allowing user interaction with the **AcceptInput()** method. This menu will be displayed until the user either selects a menu option or clicks in the background. A popup menu in a window is shown in figure 8-17.

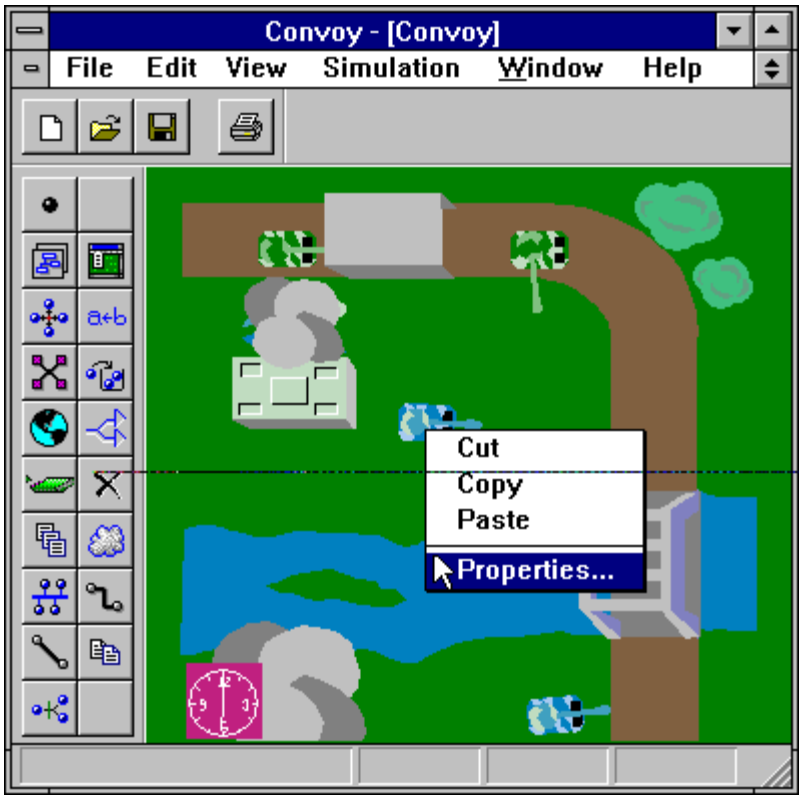


Figure 8-17. Pop-Up Menu Example

Here is an example of how to use a popup menu:

```
   { override ImageObj BeSelected() method }
ASK METHOD BeSelected;
VAR
   popupMenu : PopupMenuObj;
   cutItem, copyItem, pasteItem, pickedItem : MenuItemObj;

BEGIN
  IF window.Button > 1  { only respond to right mouse button }
   NEW(popupMenu);
   NEW(cutItem);     NEW(copyItem);   NEW(pasteItem);

      { set the labels appearing in the menu; build the
      menu }
   ASK cutItem TO SetLabel("Cut");
   ASK copyItem TO SetLabel("Copy");
```

```
    ASK pasteItem TO SetLabel("Paste");
    ASK popupMenu TO AddChild(cutItem, "cut", 0);
    ASK popupMenu TO AddChild(copyItem, "copy", 0);
    ASK popupMenu TO AddChild(pasteItem, "paste", 0);

    ASK window TO AddGraphic(popupMenu);
    pickedItem := ASK popupMenu TO AcceptInput();
    CASE pickedItem.ReferenceName
         WHEN "cut" :   ASK SELF TO Cut;
         WHEN "copy" : ASK SELF TO Copy;
         WHEN "delete" : DISPOSE(SELF);
    END CASE;

   DISPOSE(popupMenu);
   END IF;
END METHOD;
```

8.24 Palettes, Pallette Buttons, Palette Separators

Palettes (**PaletteObj**) can be created and attached to the sides of a SIMGRAPHICS II window. A palette contains rows and columns of selectable square *palette buttons* (**PaletteButtonObj**). Each palette button shows a bitmap icon on its face. On MS Windows systems, palettes can be *dockable*. At runtime you can reattach a dockable palette to a different side of its window. Palettes not docked to any edge of the window are called *floating* and behave like modeless dialog boxes. Figure 8-18 shows a window containing two palettes docked on the top and left, a menu bar and a status bar.

A palette can be created in SIMDRAW or programmatically. To use a palette, define a new type of object derived from **PaletteObj** and override the **BeSelected()** method. This method will be invoked whenever the user presses one of the palette buttons. The **LastPicked** field will point to the last palette button selected.

To receive palette selections while a simulation is running do the following:

1. Create an instance of your own palette object:

   ```
   NEW(MyPalette);
   ```

2. Ask the palette to load its description from the library:

   ```
   ASK MyPalette TO LoadFromLibrary(lib,
   "PaletteLibName");
   ```

3. Attach the palette to the window:

   ```
   ASK window TO AddGraphic(MyPalette);
   ```

4. Draw the window (or the palette) to display the palette:

   ```
   ASK window TO Draw;
   ```

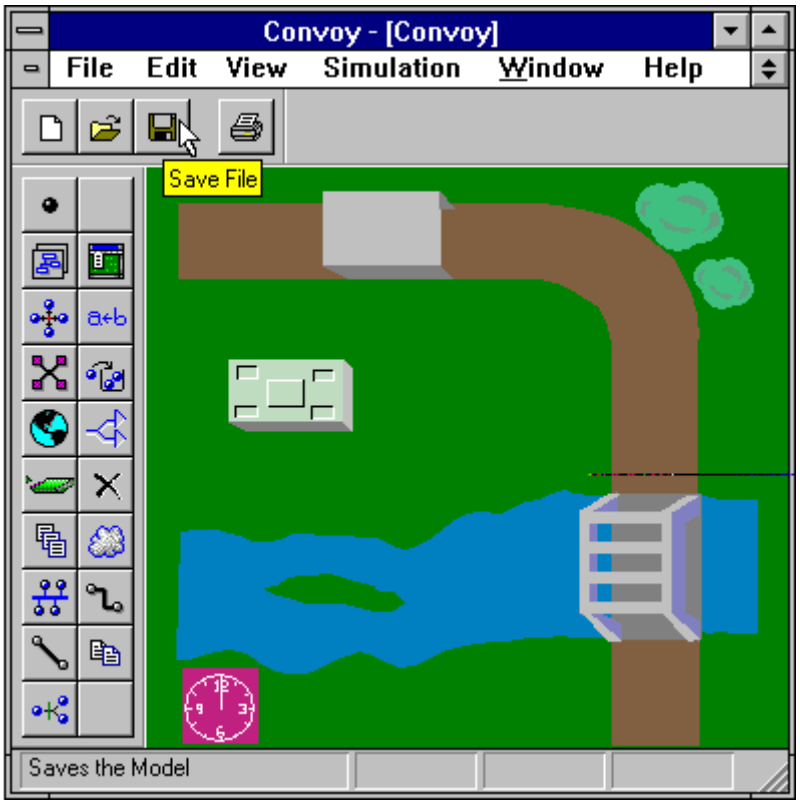


Figure 8-18. Palette Example

PALETTE EXAMPLE PROGRAM

```
MAIN MODULE Example8;
  { Simple Palette }

FROM Window  IMPORT WindowObj;
FROM GPalet  IMPORT PaletteObj;
FROM GProcs  IMPORT HandleEvents;
FROM Graphic IMPORT GraphicLibObj;

TYPE
  MyPaletteObj = OBJECT(PaletteObj)
   OVERRIDE
      ASK METHOD BeSelected;
  END OBJECT;

VAR
  myPalette : MyPaletteObj;
  window : WindowObj;
  library : GraphicLibObj;
```

```
  done : BOOLEAN;

OBJECT MyPaletteObj;
  ASK METHOD BeSelected;
  BEGIN
    CASE ASK LastPicked ReferenceName
        WHEN “cut” : OUTPUT(“Cut operation selected”);
        WHEN “copy” : OUTPUT(“Copy operation selected”);
        WHEN “paste” : OUTPUT(“Paste operation
        selected”);
        WHEN “quit” : done := TRUE;
    END CASE;
  END METHOD;
END OBJECT;

BEGIN
    { Init graphics and create window }
  NEW(window);
  ASK window TO Draw;

    {  load description of palette from the library file
created by SIMDRAW }
  NEW(library);
  ASK library TO ReadFromFile(“Example8.sg2”);
  NEW(myPalette);
  ASK myPalette TO LoadFromLibrary(library, “Palette”);

    { Add palette to the window and display it. }
  ASK window TO AddGraphic(myPalette);
  ASK myPalette TO Draw;

    { Get input from the palette }
  WHILE NOT(done)
    HandleEvents(TRUE);
  END WHILE;
END MODULE.
```

Appendices

Appendix A: Common Pitfalls

The following is a list of DOs and DON'Ts to keep in mind when programming a SIMGRAPHICS II application. Violating these rules may result in crashes which are difficult to diagnose.

DO NOT:

Attempt to draw a graphical object that is not attached to an image tree or window.

Attempt to draw a control that is not attached to a control tree or window.

Set an attribute of an ancestor, and then ask one of its descendants in the tree to **`Draw`** without first asking the ancestor to **`Draw`**. The results of this type of operation are ill-defined. However, it is OK to set an attribute of a child and then ask its parent to **`Draw`**.

Set the **`RotationSpeed`** or other speed methods of dynamic image objects to extremely high values such as 999999999.9 or a floating point exception call will be generated.

Set any of the attributes of a graphical object directly. Always invoke the correct **`Set`** method to set any attribute.

DO:

Remember to invoke the INHERITED behavior of any graphical object's method you have overridden, especially the **`ObjInit`** method.

Remember that the default behavior of the **`BeSelected`** method is to invoke an object's parent's **`BeSelected`** method.

Remember that when a graphical object is disposed of, all of the child objects in its list are automatically disposed of.

Appendix B: SIMGRAPHICS II - 3D

B.1 Lights, Cameras, Action!!

Note: 3-D is presently available on Windows NT Revision 3.51 or higher.

SIMGRAPHICS II - 3D is capable of real-time 3-D solids animation with multiple light sources and cameras. It takes advantage of 3-D hardware accelerators to animate 3-D simulation objects.

The interface to 3-D graphics is similar to the 2-D interface. 3-D image objects provide roughly the same capabilities as their 2-D counterparts. Primitives including polygons, polylines, text, and triangular meshes can also be created within a program and added to a 3-D window.

Camera objects provide an intuitive interface to viewing. They can be positioned and aimed at objects or in specified directions. In addition, cameras can be attached to 3-D images and move with their parent.

B.2 Building a 3-D Model

Constructing a simulation using 3-D is similar to building it using 2-D graphics. Like the 2-D library, the 3-D interface is provided through objects. You must create 3-D objects, add at least one light to make the objects visible, and add a camera to view the scene. The following is a summary of the steps and objects involved in creating a 3-D simulation model:

1. Create objects for background and animation using SIMDRAW in 3-D mode.
2. Create a 3-D window object. **`Window3dObj`**
3. Create a library object and load the file created by SIMDRAW:

   `GraphicLibObj`
4. Create static and dynamic 3-D images and load their graphic representations from the library. Add them to the 3-D window.

   `Image3dObj, DynImage3dObj`
5. Create and position a light to illuminate the 3-D world. **`LightObj`**
6. Create and position a camera and direct it to look at a location or object in the 3-D world. Add it to the 3-D window. **`CameraObj`**
7. Ask the 3-D window to draw.
8. Start the simulation.

To animate the 3-D objects, give them a speed, and then ask them to:

1. Move to a specific coordinate:

```
ASK obj TO MoveTo( x, y, z)
```

2. Follow a path defined by an array of 3-D points

```
ASK obj TO FollowPath(points)
```

3. Rotate to an angle:

```
ASK obj TO RotateTo(xangle, yangle, zangle )
```

4. Scale to a size:

```
ASK obj TO ScaleTo( xsize, ysize, zsize )
```

B.3 Objects Used

Window3dObj

The 3-D Window object acts as a container for all 3-D images, dynamic images and primitives. The 3-D window is created, positioned and sized identically to the 2-D window object. The rendering mode for all graphics drawn in the window can be set to wireframe, solid, or solid with shading modes. The 3-D window object also controls the lighting equation by setting the amount of ambient, diffuse, and specular reflection. The defaults for the lighting equation work for most situations.

Note: Not all lighting and shading options are supported on all 3-D platforms.

Image3dObj and DynImage3dObj

3-D Images are similar to the 2-D images. They can be positioned, rotated, and scaled. They can also be instructed to aim at a point, or to track another 3-D object. Like 2-D images, 3-D images can be hierarchical. A 3-D image can contain sub-images which in turn contain sub-images. For example, a 3-D tank can have a 3-D turret attached to it as a separate object, which moves, rotates, and scales along with the tank. Dynamic 3-D images can move to a 3-D point in the modelling space, scale to any size, and rotate to any angle over simulation time. It can also follow a path defined by an array of 3-D points.

CameraObj

A least one camera must be **Cue**d to view the 3-D world. Cameras are positioned in 3-D space and are oriented either by rotation, or by aiming at a location. Cameras can also **Track** 3-D objects. A tracking camera will automatically adjust itself to point at the tracked object whenever the tracked object changes location. Cameras can be attached to any moving or rotating 3-D object, enabling you to see from the object's vantage point.

Any number of cameras can simultaneously view a scene. The output of each camera goes to a section of the 3-D window described by the camera's "view port."

Cameras also have a depth of field and can zoom in and out. The depth of field for the camera controls the near and far viewing limits. Objects and parts of objects which lie closer than the near distance or greater than the far distance are not displayed.

The camera's zoom factor controls the viewing angle. You may zoom in or out. Zooming in decreases the viewing angle and enlarges the scene. Zooming out decreases the vie wing angle and reduces the scene.

`LightObj`

Light objects can be created to help illuminate a scene. Light objects can be positioned and oriented in 3-D space. Any number of lights up to the maximum allowed by the 3-D hardware can be created. 3-D images are automatically shaded using the lights' color and position.

The three different types of lights available are directional, positional, and spot lights.

Directional lights are similar to sunlight and are often the only lights used to illuminate a 3-D scene. Directional lights shine on all objects with the same intensity regardless of the object's location. The light's position and its aim point determine a vector which repr esents the light's direction.

Positional lights are similar to uncovered light bulbs. They illuminate in all directions with intensity diminishing with distance. A positional light will illuminate an object di fferently depending upon the position of the object and the light source. The attenuation field determines how the light will fade with distance.

Spot lights are similar to a covered lights. They produce cones of illumination which diminish with distance. The spot light's position and aim point determine the line of maximum illumination. Its spread angle determines the width of the illumination cone and its concentration determines how the light varies as distance is increased from the center of the illumination cone. Attenuation determines how the light fades with distance. Spot lights can also track 3-D objects to keep maximum illumination focussed on the tracked object.

Lights can be oriented through rotation, or can be asked to aim at a point. Like cameras, a light can be attached to a hierarchy of 3-D images, and automatically moves and rotates with its parent.

B.4 Combining 2-D and 3-D Graphics

One application can have both 2-D and 3-D graphics windows, but only 2-D graphic objects can be displayed in the 2-D window and only 3-D objects can be displayed in the 3-D window.

B.5 3-D Primitives

There are three 3-D primitives available: **`Polyline3dObj`**, **`Polygon3dObj`**, and **`TriangularMeshObj`**. SIMDRAW builds objects using only **`Polyline3dObj`** and **`Polygon3dObj`**'s. However, all three primitives can be created programmatically.

`Polyline3dObj`

The 3-D polyline object allows a line with an arbitrary number of line segments to be drawn. The number of points that a polyline can contain is limited only by the display hardware.

`Polygon3dObj`

The 3-D polygon primitive allows a polygon with an arbitrary number of vertices to be drawn. The points in the polygon must be planar. (i.e all points should line up in a plane.) The **`polygon3d`** object can be used build complex objects by adding polygons to a 3-D image or dynamic image objects.

`TriangularMeshObj`

The triangular mesh primitive is the most efficient primitive in terms of rendering speed and memory usage. Each new point in the points array for the triangular mesh defines a new triangle using itself and the two points immediately preceding it in the array. See figure B-1.

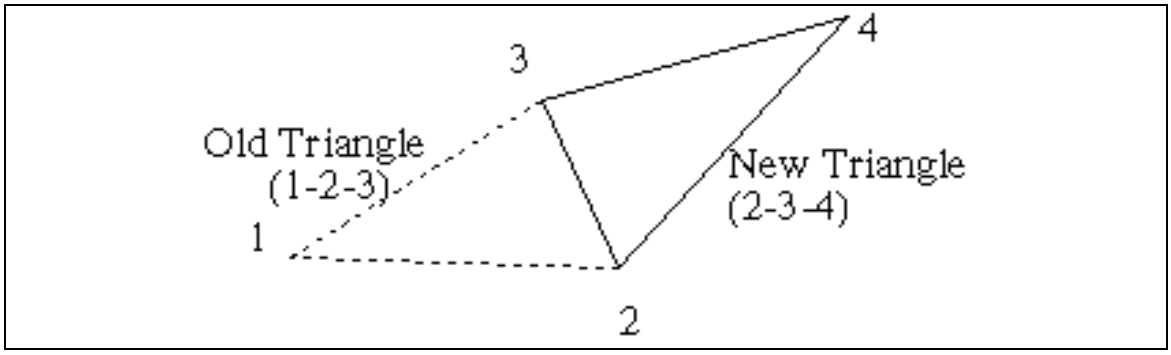


Figure B-1. New Triangle Formed by Point 4 in the Triangular Mesh Array

B.6 Tips for 3-D Simulation

Setting up a 3-D simulation model is a little more difficult than it is for two dimensions, because there are cameras and lights to consider. It is easy to overlook a detail and not get anything to appear on the screen. If this happens make sure a **`CameraObj`** has been created and added to the either the window or the graphics hierarchy. The camera's depth of field must be set so that the objects being viewed are between the near and far clipping

planes (see figure B-2). Also make sure the camera is pointing toward the objects you are viewing.

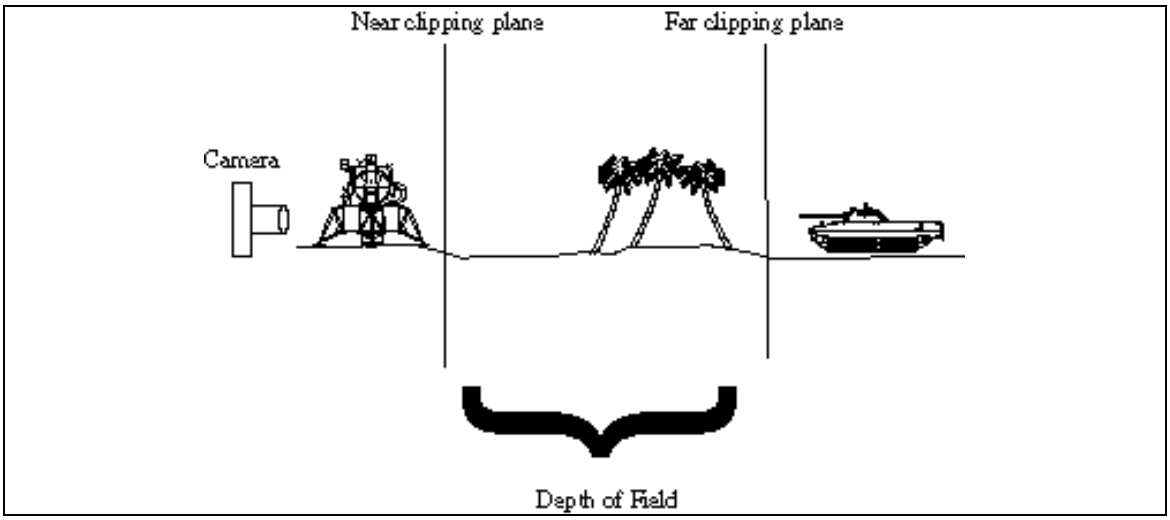


Figure B-2. The Camera Can See the Palm Trees, but not the Lunar Module

Appendix C: Canvas and System Cursors

A window can optionally have a system or canvas *cursor*. A canvas cursor is an object derived from **ImageObj** that tracks with the mouse. A system cursor can be the normal system pointer, or a busy cursor such as an hourglass or watch.

C.1 Using Canvas Cursors

Since a canvas cursor is an object derived from **ImageObj**, any object can be used as a cursor. If a window's **MoveMonitoring** flag is set, this cursor will automatically move with the mouse pointer. If the **MonitoringClicks** flag is set the cursor will appear at the pointer position every time the mouse button is pressed. The **SetCursor** method is used to set the cursor of a window.

Any object derived from **ImageObj** can be used as a cursor. However, the **Cursor** module contains several predefined cursor objects derived from **PolylineObj** that provide for efficient rubberband cursors. The line cursor is the simplest of them. It draws a line from an anchor point to the current cursor position. The following code sets the anchor point for the line cursor and initiates tracking:

```
FROM Cursor IMPORT LineCursorObj;
...
VAR linecursor : LineCursorObj;
...
NEW(linecursor);
ASK linecursor TO SetAnchor(WorldXlo, WorldYlo);
ASK window TO SetCursor(linecursor);
...
```

Note: User-defined cursors must be attached to a window or image tree before being set as the window's cursor. However, the special predefined cursors should NOT be attached to a window or image tree. This is because the special cursors bypass the standard graphics library to perform drawing and therefore can not be safely added to an image tree.

If you do not want the **ImageObj** or rubber band cursor to be displayed exactly at the current cursor position, you can specify an x,y offset from the mouse position to where it will be drawn. This is done using the **SetCursorOffset** method. The following code fragment causes any cursor attached to the window to be drawn at (pointer x coordinate) + 5000.0 , (pointer y coordinate) + 2000.0 .

```
...
ASK window TO SetCursorOffset(5000.0, 2000.0);
...
```

There are five cursor objects which provide efficient rubber band cursors. These are **LineCursorObj**, **RectCursorObj**, **FixedAspCursorObj**, **FixedCursorObj** and **DualLineCursorObj**. Each of these has an anchor point from which the base of the rubber band line is drawn. The **SetAnchor** method sets the anchor point. Here is a description of each:

LineCursorObj—The **LineCursorObj** is used to draw a rubber band line from an anchor point to the current mouse position.

RectCursorObj—The **RectCursorObj** is used to draw a rubber band rectangle from an anchor point to the current mouse position.

FixedAspCursorObj—The **FixedAspCursorObj** is used to draw a rubber band square or other rectangle with a fixed aspect ratio from the anchor point to the current mouse position. The aspect ratio of the cursor remains constant by using the maximum of the horizontal or vertical distances from the anchor point as the dimension for the side of the rectangle. The ratio is specified using the **SetRatio** ASK method. The ratio is given as width / height.

```
...
ASK fixedcursor TO SetRatio(2.0);
...
```

FixedCursorObj—The **FixedCursorObj** is used to draw a fixed rectangle with its lower left corner placed at the current mouse position. The size of the rectangle is specified using the **SetSize** method:

```
...
ASK fixedcursor TO SetSize(width, height);
...
```

DualLineCursorObj—The **DualLineCursorObj** is used to draw two rubber band lines from two anchor points to the current pointer position. The second anchor point is specified using the **SetDualAnchor** method:

```
...
ASK duallinecursor TO SetDualAnchor(x, y);
...
```

C.2 Using System Cursors

An hour glass, stop watch, etc., can be displayed instead of the system cursor. See the modules **Form** and **Window**. The following field and method are new in both the **WindowObj** and **DialogBoxObj** objects:

```
SysCursor : SysCursorType ; { Current system cursor
                              type }
```

```
ASK METHOD SetSysCursor (IN cursorType :SysCursorType);

{  Changes the system cursor used when the pointer is
within this window or dialogbox.  See GTypes for a list
of supported cursor types. }
```

To use the system cursor, ask each window or dialog box to **SetSysCursor**:

```
FROM GTypes IMPORT ALL SysCursorType ;
...
ASK win TO SetSysCursor(BusyCursor) ;
ASK db  TO SetSysCursor(BusyCursor) ;
...
ASK win TO SetSysCursor(NormalCursor) ;
ASK db  TO SetSysCursor(NormalCursor) ;
```

Appendix D: Creating Images at Runtime

Although images can be created within SIMDRAW, a user may occasionally want to create them dynamically at runtime. This can be done for all primitives. Doing this involves performing a **NEW** on the object, setting its attributes, and then drawing the object. All attributes of a visible object can be reset at runtime with changes being seen at the time a **Draw** is done.

Creating primitives dynamically is quite simple. It involves setting the fields of the primitive, and then adding it to a window or image.

Note: Fields of a primitive are always initialized to an appropriate default value.

D.1 Objects

PrimitiveVObj

All primitives are derived from this object. Primitives include polylines, polygons polymarks, text, arcs, circles and sectors. Primitives are not hierarchical: you should not ask a primitive to **AddGraphic** another primitive or **ImageObj**. The only attribute of a **PrimitiveVObj** is its Points array. The Points field is of type **PointArrayType** which is a one dimensional array of **PointType** records. (These types are found in the **GTypes** module.) The points of a primitive define what its shape will be. The **SetPoints** method sets the points of a primitive. The number of points defining the primitive is determined by the size of the given points array. All primitives are drawn using the color attributes derived from **ImageObj**. Primitives cannot hold other primitives. Do not try to add to polygons, polylines, arcs, circles, snapshots, etc.

PolylineObj

A polyline; is a line that connects the points of the Points array. One of the attributes used in drawing a line is its **Style**, which determines the line style to be used in drawing. This attribute's type is **LineStyleType**, which is defined in the **GTypes** module and can take on the following values:

```
LineStyleType = (SolidLine, LongDashedLine, DottedLine,
DashDottedLine, DashedLine, DashDotDottedLine,
ShortDashedLine, AlternateLine);
```

Two more attributes of a polyline are its **Width** and **PctWidth**. The **Width** attribute determines the line's width in terms of real world x,y coordinate space. If a line's **Width** is set, then the line width will be scaled whenever the line or one of its ancestor images is scaled. The **PctWidth** attribute determines a line's width in terms of percentage of window size. When you set the **PctWidth** attribute of a line, its width is NOT scaled when the line is scaled or an ancestor is scaled; the line will stay the same width regardless of what its size is.

ArcObj

Since **ArcObj** and **PolylineObj** are derived from a common object **LineVObj**, an arc contains the same style and width attributes as a polyline. An arc is defined by 3 points in the points array. The first point identifies the center of the arc. The second point, a point on the circumference, is the beginning of the arc. The arc is drawn counterclockwise to the third point (also on the circumference). Any points after the third are ignored.

PolygonObj

The real world points in the points array describe the boundaries of an area to be filled. Fill primitives (polygons, sectors, and circles) have a **Style** attribute identifying the fill style used in drawing. This attribute is of type **FillStyleType** which is defined in the **GTypes** module as follows:

```
FillStyleType = (HollowFill, SolidFill,
NarrowDiagonalFill,MediumDiagonalFill, WideDiagonalFill,
NarrowCrosshatchFill, MediumCrosshatchFill,
WideCrosshatchFill);
```

CircleObj

A circle is described by the first two points in the points array. The first point is the center of the circle, and the second is any point on its circumference.

SectorObj

A sector is a sector of a circle. It is a fill primitive like a circle, but is described in the same way as an arc.

TextObj

A **TextObj** primitive is a graphical display of a text string. It has attributes describing its alignment, size, font, and the string value displayed. The font attribute is the font in which the text value will be drawn. The following fonts are available:

```
TextFontType = (SystemFont, SimpleFont, RomanFont,
BoldRomanFont, ItalicFont, ScriptFont, GreekFont,
GothicFont, SystemText);
```

D.2 System Text

Previously only vector-based fonts could be displayed in a graphics window. System text provides access to all system fonts available on the underlying graphics system. An optional translation file provides the capability to remap fonts when the application is moved to a different graphical environment. See the libraary module Text. These are the **TextObj** fields and methods relating to system text:

```
Family : STRING ; { System font family }
 Weight : INTEGER ;{ System font weight, 0 = lightest,
                     50 = normal; 100 = boldest }
 Size : INTEGER ;  { System font size; in points }
 Slant : INTEGER ; { System font slant; 0 = normal; 1 =
                     italics }
 SysFontStatus : INTEGER ; { 0 = bad system font spec;
                             1 = found font }

 PROCEDURE LoadSFontFile (IN filename: STRING)
                           : FileStatusType ;
 {  Attempts to load the specified file of system text font
 name translations.  Each entry in this file consists of a
 pair of double-quoted (") strings on each line - the
 first is a name to be translated, the second is the
 translation.  The intent is that font names for one
 platform will be translated to an appropriate name on
 another.  Note that there are built-in translations; any
 translations loaded by LoadSFontFile will be searched
 before the built-in translations. }

 ASK METHOD SetSysFont(IN Family : STRING ;
     IN Size   : INTEGER ; IN Weight : INTEGER ;
     IN Slant  : INTEGER  ) ;
 {  Sets the system text font to use if the Font field is
 SystemText.  An appropriate font specification can be
 obtained from the Form module FontDialogBoxObj object.}

 ASK METHOD SetNoFontError(IN ignoreSysFontErrors
                           : BOOLEAN);
 {  If ignoreSysFontErrors is TRUE, the 'SysFontStatus'
 variable will be set without a runtime error being
 generated whenever the specified font can't be loaded.
 Default is TRUE. }
```

D.3 Using System Text

Here are the steps involved in using system text:

1. Set the font to **`SystemText`** (**`SetFont`**).

2. Set the font alignments (**`SetAlignment`**). This determines the position of the text relative to the reference point and translation.

3. **`SetTranslation`** to move the text to the desired position. Note that the text will be positioned relative to the specified point according to the alignments selected.

4. Set the **`Family`**, **`Size`**, **`Weight`** and **`Slant`** fields using **`SetSysFont`**. The most direct way to get these is by using the system font browser dialog box. Note that, by default, if the specified font cannot be found, a default system font will be used. Using method **`SetNoFontError`**, this behavior can be changed so that an error will occur if the font cannot be found.

EXAMPLE:

```
NEW(text);
ASK text TO SetFont(SystemText)
ASK text TO SetTranslation(6000.0, 8000.0)
ASK text TO SetAlignment(HorizLeft, VertMiddle)
ASK text TO SetSysFont("Times Roman", 12, 100,0);
```

D.4 Portability Issues

1. For Unix/X Windows platforms the Family field can be a complete X11 font name, as might be obtained from the **`xfontsel`** or **`xlsfonts`** programs usually included with the X Windows release. In this case, the size, weight and slant fields will be ignored. If only the foundry and font family names are used (separated by a hyphen), the size, slant and weight fields will be used to construct a standard X11 font specification. The following are all possible Family strings for X11 platforms:

   ```
   "fixed"
   "adobe-new century schoolbook"
   "-adobe-times-medium-i-*-*-*-140-*-*-*-*-iso8859-1"
   ```

2. Any rotation can be specified; the text will be rendered at the nearest increment of PI/2.

3. The size of system text is determined by the font selected, and so, unlike stroked text, does not change with resizing of windows or setting the scaling factor of

ancestors in the image tree. Therefore, when the window is resized, the bounding box (in Normalized Display Coordinates or NDCs) will change. See the note on bounding boxes above.

4. To enhance portability, a font translation file can be created, which provides aliases for the font's Family field. This file is read in using the Text module procedure **`LoadSFontFile`**. This is a text file consisting of pairs of strings, enclosed in double quotes and separated by white space; the first is a string that may be specified as the Family field of a **`TextObj`**, the second is a translation that will be substituted by the graphics library when searching for fonts to use to render the text. Note that the aliases are substituted and searched for until a valid font is found, and there can more than one translation for a given alias, so that a single translation file can be used on more than one platform. For example, loading a translation file containing these lines:

   ```
   "Times" "adobe-times"
   "Times" "Times Roman"
   ```

   and setting Family to "Times" will match an Adobe Times font on an X11 platform, and a Times Roman font on Microsoft Windows.

The **`SetAlignment`** method specifies how the text is to be aligned upon its position. Text can be aligned horizontally and vertically. Vertical text alignment is based on a character cell which extends both above and below the actual character. Horizontal alignment is done with respect to the length of the entire string. The alignments available are:

```
TextHorizType = (HorizLeft, HorizCentered, HorizRight);
TextVertType = (VertBottom, VertMiddle, VertTop,
VertCellBottom, VertCellTop);
```

Text sizing can be specified in one of two ways. You can use the **`SetSize`** method to specify a “box” to fit the text string into. This box's width and height are given in real world coordinate space. Text with sizing specified in this manner will be scaled in size whenever the object is scaled. The **`SetPctHeight`** method is another way of giving text size. This method specifies text height as a percentage of window height. Text with this type of sizing is not scaled; therefore it will always appear the same size.

Text can be rotated using the **`SetRotation`** method.

D.5 Markers

`PolymarkObj`

A polymarker is a set of small graphical markers drawn at every point in the points array. The markers have both a size and style. The style of a marker identifies what shape the marker has. The following styles are available:

```
MarkStyleType = (DotMark, CrossMark, StarMark,
SquareMark, XMark, DiamondMark);
```

Marker size is specified much the same way as text size. The **SetSize** specifies a size in real world x,y coordinates. This method will scale marker size with the object it's attached to. **SetPctSize** method gives a percentage of window height. After this method is called, a marker will not scale in size. These methods have the following format:

```
ASK METHOD SetSize(IN heightOfMarker : REAL);
ASK METHOD SetPctSize(IN pctOfWindow : PctType);
   { 0.0 to 100.0 }
```

D.6 SnapShot Object

The **SnapShotObj** is a bitmapped graphic primitive that can be attached to an image like any other primitive. The file containing bitmap data (files with extensions " .xwd" on XWindows and ".bmp" on MS Windows) is given using the **SetFile** method. The first point in the **Points** array field is the lower left hand corner of the bitmap and the second point is its upper left hand corner. If the bitmap is **non-scalable** these points will be defined automatically by SIMGRAPHICS II. For **scalable** bitmaps the programmer must define these points. See chapter 5 for more information on using the **SnapShotObj**.

D.7 Example Program

The following example creates a small "cart" dynamically:

```
MAIN MODULE Example7;

{ This module creates a 2 level 'Cart' and displays it on
the screen. Its first wheel is then removed, then its
second wheel is removed. }

FROM GTypes IMPORT PointArrayType,
 FillStyleType(SolidFill, MediumDiagonalFill),
 ColorType(Green, Yellow);
FROM Window IMPORT WindowObj;
FROM Fill IMPORT CircleObj, PolygonObj;
FROM Image IMPORT ImageObj;

VAR
   window : WindowObj;
   wheel1, wheel2 : CircleObj;
   cart : ImageObj;
   body : PolygonObj;
   wheelpoints : PointArrayType;
   bodypoints : PointArrayType;

BEGIN
     { bring up the window }
   NEW(window);
   ASK window TO Draw();

     { assign points for all wheels }
   NEW(wheelpoints, 0..1);
   wheelpoints[0].x := 0.0;
   wheelpoints[0].y := 0.0;
   wheelpoints[1].x := 1500.0;
   wheelpoints[1].y := 0.0;

     { assign points for body image }
   NEW(bodypoints, 0..3);
   bodypoints[0].x := 0.0;
   bodypoints[0].y := 0.0;
   bodypoints[1].x := 10000.0;
```

```
    bodypoints[1].y := 0.0;
    bodypoints[2].x := 10000.0;
    bodypoints[2].y := 6000.0;
    bodypoints[3].x := 0.0;
    bodypoints[3].y := 6000.0;

      { create a body primitive }
    NEW(body);
    ASK body TO SetPoints(bodypoints);
    ASK body TO SetColor(Yellow);
    ASK body TO SetStyle(SolidFill);

      { create a wheel primitive }
    NEW(wheel1);
    ASK wheel1 TO SetColor(Green);
    ASK wheel1 TO SetStyle(MediumDiagonalFill);
    ASK wheel1 TO SetPoints(wheelpoints);

      { make a copy of it }
    wheel2 := ASK wheel1 TO Copy();

      { now build the tree, and add it to the root }
    NEW(cart);
    ASK window TO AddGraphic(cart);
    ASK cart TO AddGraphic(body);
    ASK cart TO AddGraphic(wheel1);
    ASK cart TO AddGraphic(wheel2);

      { set position of wheels in }
      { relation to their parent. }
    ASK wheel1 TO SetTranslation(2000.0, 0.0);
    ASK wheel2 TO SetTranslation(8000.0, 0.0);

       { Draw the cart }
    ASK cart TO DisplayAt(16383.0, 16383.0);

    LOOP
    END LOOP;
  END MODULE.
```

Appendix E: Animation Speed Optimization

SIMGRAPHICS II is designed for fast and smooth animation. It contains some features that can be used to obtain even faster animation. Certain rules can also be followed to obtain maximum performance.

E.1 Real-time Animation Mode

A real-time animation mode adds faster, smoother animation during a simulation. The time taken for each animation movement is compared to the elapsed wall-clock time ; if the simulation begins to fall behind, graphical updates are skipped as required to keep up. To invoke real-time animation:

```
FROM Dynamic IMPORT RealTimeAnimation;
BEGIN
     RealTimeAnimation := TRUE;
     . . .
     StartSimulation;
     . . .
```

E.2 SetSnapShot

A snapshot can be taken of an image and used for animation. This snapshot can be drawn to the window faster than the vector representation of the image . A complicated snapshot is drawn in the same time as a simple snapshot. The **SetSnapShot** method marks an image so that snapshots are taken automatically.

```
...
ASK truck TO SetSnapShot(TRUE);
...
```

SnapShot should only be used on images whose appearance remains static. An image that is continually scaled and rotated or an image whose children are continually modified or repositioned should not set snapshot. (When the description of an image changes, the library must throw away its snapshot and construct a new one. This can be expensive if the image's description is continually changing.)

E.3 SetRedrawable

When the **Redrawable** flag of an image is set, it will be automatically repainted if another image on top of it is erased. This is why you need not worry about redrawing images that have been “run-over” by another image. Images are created with the **Redrawable** flag set to **TRUE**, but this flag can be cleared using the **SetRedrawable** method. Images that will never be run over by another image can have the **Redrawable** flag set to **FALSE** and some animation speed improvement may be seen. For example,

suppose you had an image named **shoreLine** that could never be obscured by another image:

```
...
ASK shoreLine TO SetRedrawable(FALSE);
...
```

This flag could also be cleared if you *wanted* to see an image be eaten away by another image.

E.4 Exclusive OR Drawing Mode

SIMGRAPHICS II provides an "exclusive-or" drawing mode. When using this mode the graphics library does not have to "repair" parts of the scene that are run over by a moving image, thus leading to faster animation. This mode is recommended only for simple scenes and is best suited for images that are composed of *polylines*. The **WindowObj** method **SetFastDraw** sets the exclusive-or drawing mode. Note that the mode applies to ALL objects in the window.

E.5 Miscellaneous Tips on Faster Animation

Remember that the time it takes to draw an image is proportional to the size of the bounding box (the smallest rectangle that encloses the image). A big image takes much longer to draw than a small one. Therefore, if you had an image whose children were positioned far apart from each other, this image would take a long time to draw, since the bounding box of the image is huge. In this case it is advisable to draw the children of the image instead of the image itself. For example, suppose **bigImage** was composed of the two images **farChild1** and **farChild2** and that these children were translated to opposite corners of the window. The code:

```
ASK farChild1 TO Draw();
    { farChild1 at (0.0, 0.0) }
ASK farChild2 TO Draw();
    { farChild2 at (32767.0, 32767.0) }
```

is preferable to:

```
ASK bigImage TO Draw();
```

E.6 Command Line Options

Two command line options may be passed to SIMGRAPHICS II executables:

-nograp This option causes your executable to run without displaying any graphics, including user input dialog boxes and menu bars.

`-noimage` This option causes the executable to run without displaying animation, but it will display dialog boxes and menu bars to allow user input.

In both cases, **`TimeScale`** is set to run the simulation as fast as possible.

Appendix F: Complete Solar System Example

Figure F-1. Solar System

```
MAIN MODULE Example3;

{   Example3  -- Complete model of solar system.

  The following example graphically models the solar system
and the rotating planets.  Planets spin on their axis while
rotating around the sun.   Every earth hour, the planet
positions and rotational spins are updated. }
FROM GTypes IMPORT WorldXhi, WorldYhi;
FROM Graphic IMPORT GraphicLibObj;
FROM Window IMPORT WindowObj;
FROM Icon IMPORT IconVObj;
FROM Image IMPORT ImageObj;

CONST
     MercuryYear = 87.784;
       { # earth days in a mercury year }
     VenusYear = 224.013;
       { # earth days in a venus year }
     EarthYear = 365.25;
       { # earth days in an earth year }
```

```
      MarsYear = 685.15;
        { # earth days in a mars year }

      MercuryDay = 1416.0;
        { # earth hours in a mercury day }
      VenusDay = 6000.0;
        { # earth hours in a venus day }
      EarthDay = 24.0;
        { # earth hours in an earth day }
      MarsDay = 24.616;
        { # earth hours in a mars day }

      MercDist = 3.217;
        { mercury light minutes from sun }
      VenusDist = 6.011;
        { venus light minutes from sun }
      EarthDist = 8.310;
        { earth light minutes from sun }
      MarsDist = 12.662;
        { mars light minutes from sun }

      SystSize = 15.0;
        { radius in light minutes of this model }

VAR
      library : GraphicLibObj;
      window : WindowObj;
      solarSystem : ImageObj;
      sun, mercury, venus, earth, mars : ImageObj;
      mercuryOrbit, venusOrbit, earthOrbit,
      marsOrbit : ImageObj;
      mercuryRot, venusRot, earthRot, marsRot : INTEGER;
      mercuryOrbitRot, venusOrbitRot, earthOrbitRot,
      marsOrbitRot : INTEGER;
      earthHours : REAL;

BEGIN
        { create a window for the solar system, and
          bring it up on the screen }
    NEW(window);
    ASK window TO Draw();
```

```
    { create a library and have it read the saved
      library "Example3.sg2" }
NEW(library);
ASK library TO ReadFromFile("Example3.sg2");
    { get planets from the library. Assume they have
      been saved using the names "SUN", "MERCURY",
     "VENUS", "EARTH" and "MARS" in SIMDRAW }

NEW(sun);
NEW(mercury);
NEW(venus);
NEW(earth);
ASK sun     TO LoadFromLibrary(library, "SUN");
ASK mercury TO LoadFromLibrary(library, "MERCURY");
ASK venus   TO LoadFromLibrary(library, "VENUS");
ASK earth   TO LoadFromLibrary(library, "EARTH");
ASK mars    TO LoadFromLibrary(library, "MARS");
    { Build the solar system.  Special 'Orbit
      images' are added as parents of the planets.
      Rotating these will cause the planets to
      rotate around the sun.  Rotating the planets
      themselves causes them to spin on their axes }

NEW(solarSystem);
    { create the solar system }
NEW(mercuryOrbit);
    { create new orbit images }
NEW(venusOrbit);
NEW(earthOrbit);
NEW(marsOrbit);
ASK solarSystem TO AddGraphic(mercuryOrbit);
    { add the orbit images to the }
ASK solarSystem TO AddGraphic(venusOrbit);
    { solar system }
ASK solarSystem TO AddGraphic(earthOrbit);
ASK solarSystem TO AddGraphic(marsOrbit);
ASK solarSystem TO AddGraphic(sun);
ASK mercuryOrbit TO AddGraphic(mercury);
    { add the planets to their orbit }
ASK venusOrbit TO AddGraphic(venus);
```

```
    { images }
ASK earthOrbit TO AddGraphic(earth);
ASK marsOrbit TO AddGraphic(mars);
    { add solar system to the universe }
ASK universe TO AddGraphic(solarSystem);
    { Now set the world coordinate system of the
      solar system.  Positioning of the solar system
      is then given with respect to to the default
      world,  and positioning of planets is given
      with respect to the solar system world. }

ASK solarSystem TO SetWorld(-SystSize, -SystSize,
 SystSize, SystSize);
ASK solarSystem TO SetTranslation(WorldXhi/2.0,
 WorldYhi/2.0);
ASK mercury TO SetTranslation(MercDist, 0.0);
ASK venus TO SetTranslation(VenusDist, 0.0);
ASK earth TO SetTranslation(EarthDist, 0.0);
ASK mars TO SetTranslation(MarsDist, 0.0);

    { draw the entire solar system }
ASK solarSystem TO Draw();

    { Each earth hour, compute the amount of
      rotation of each planet and planet orbit, then
      rotate planets and orbits }
LOOP
    { compute rotations of planets }
  mercuryRot := TRUNC((earthHours / MercuryDay) *
   360.0);
  venusRot := TRUNC((earthHours / VenusDay) *
   360.0);
  earthRot := TRUNC((earthHours / EarthDay) *
   360.0);
  marsRot := TRUNC((earthHours / MarsDay) * 360.0);

    { compute rotations of orbits of planets }
  mercuryOrbitRot := TRUNC((earthHours / (EarthDay *
   MercuryYear)) * 360.0);
  venusOrbitRot := TRUNC((earthHours / (EarthDay *
   VenusYear)) * 360.0);
```

```
      earthOrbitRot := TRUNC((earthHours / (EarthDay *
       EarthYear)) * 360.0);
      marsOrbitRot := TRUNC((earthHours / (EarthDay *
       MarsYear)) * 360.0);

        { rotate the planets }
      ASK mercury TO SetRotation(mercuryRot);
      ASK venus TO SetRotation(venusRot);
      ASK earth TO SetRotation(earthRot);
      ASK mars TO SetRotation(marsRot);

       { rotate the planets around the sun }
      ASK mercuryOrbit TO Rotate(mercuryOrbitRot);
      ASK venusOrbit TO Rotate(venusOrbitRot);
      ASK earthOrbit TO Rotate(earthOrbitRot);
      ASK marsOrbit TO Rotate(marsOrbitRot);

        { update every 6 earth hours }
      earthHours := earthHours + 6.0;
   END LOOP;
END MODULE.
```

Appendix G: Utility Procedures

G. 1 Utilities

There are some procedures available to perform various functions regarding graphical objects. These procedures are found in the module **GProcs** and are listed below:

PROCEDURE HandleEvents (IN waitForEvent : BOOLEAN);

This procedure calls the low level event handler for a system. It is used to detect mouse clicks, mouse movement, image selection and form input. It is called automatically if you perform an **AcceptInput** method or are running a simulation. It must be called at least four times a second to receive mouse input from the user if **waitForEvent** is **FALSE**.

```
PROCEDURE GetRGBColor
    (IN color : ColorType;
     OUT r,g,b : PctType);
```

Given a predefined color, its red, green, and blue triple is computed and returned.

PROCEDURE ScreenRatio() : REAL;

Returns the ratio of screen width to screen height on a given system.

```
PROCEDURE Transform
    (IN sourceImage, destImage : ImageObj;
     IN sourceX, sourceY : REAL;
     OUT destX, destY : REAL);
```

This procedure transforms a point given under **sourceImage**'s world coordinate system to one given in **destImage'**s coordinate system. If **NILOBJ** is given as the source or destination image, the **DefaultWorld** is used as the coordinate system for this image. Both images must be attached to the same tree at the time this procedure is called. Referring to the solar system example of Appendix F, suppose we had a point in Earth coordinate system units that we wanted to transform into Mars coordinates:

```
...
Transform(earth, mars, earthPt.x,
          earthPt.y, marsPt.x, marsPt.y);
...
```

Or to transform a default world coordinate into Earth coordinates:

```
...
Transform(NILOBJ, earth, mouseX, mouseY,
           earthPt.x, earthPt.y);
...
```

PROCEDURE SnapShotFileExtension() : STRING

Returns the extension of a **SnapShot** (raster) file in the form `.xwd`, `.bmp`, etc. depending on which toolkit the host machine is running under.

PROCEDURE Is3DGraphicsAvailable() : BOOLEAN

Returns **TRUE** if 3-Dimensional Graphics is available. The machine must support the OpenGL Graphics library in order for SIMGRAPHICS II 3-D support to function.

PROCEDURE SetFrameTitle(IN title : STRING);

Sets the title displayed on the frame window. This procedure will reset the title displayed on the header bar of the frame window. The procedure can be called before or after the frame window has been made visible, and will automatically update the title.

PROCEDURE SetFrameIconNames(IN smallIconName, largeIconName : STRING);

Sets the icons used when the frame window is minimized. If the application contains a frame and sub-windows, this procedure will identify either the resource or bitmap file names of the icons representing the minimized application.

PROCEDURE SetFrameTranslation(IN tx, ty : PctType)

If the application contains a frame and sub-windows, this procedure will specify the initial position of the lower left hand corner of the frame window. Position is specified in "screen" coordinates (where the lower left hand corner of the computer screen is (0,0) and the upper right corner is (100,100). This procedure must be called before the first sub-window is drawn.

PROCEDURE SetFrameSize(IN width, height : PctType)

If the application contains a frame and sub-windows, this procedure will specify the initial size in "screen" coordinates ([0,100], [0,100]) of the frame window. This procedure must be called before the first sub-window is drawn.

PROCEDURE GetFrameTranslation(OUT tx, ty : PctType)

If the application contains a frame and sub-windows, this procedure will retrieve the current position of the lower left hand corner of the frame window. Position is specified in "screen" coordinates [0,100], [0,100].

PROCEDURE GetFrameSize(OUT width, height : PctType)

Gets the current size of the frame window. If a sub-windowed style application is being used, this procedure will retrieve the current size in "screen" coordinates ([0,100], [0,100]) of the framewindow.

Appendix H: Runtime Graphics Errors

GRAPHICS ERROR 200: Unexpected NIL arg to method

Make sure you have created an instance of a graphic object before adding it.

GRAPHICS ERROR 201: Replacement must involve object of same type

Only objects of the same type may be replaced.

GRAPHICS ERROR 202: Attempt to update or draw a control which has not been attached to a container control

Check to see if a control has been added to a container control such as a dialog box, menu bar, list box, or radio box before asking it to update or draw.

GRAPHICS ERROR 203: Attempt to update or draw a control whose parent control has not been drawn

The dialog box or menu bar must be drawn before attempting to draw a control it contains. The same is true for list boxes and radio boxes.

GRAPHICS ERROR 204: Attempt to attach an object to a non-container control

There are restrictions on the type of objects which can be attached. Menu bars can add menus only, menus can add menu items only, etc.

GRAPHICS ERROR 205: Attempt to attach a control which is already attached

You must remove a control before you can add it again.

GRAPHICS ERROR 206: Attempt to reset course while moving to a destination

You must stop motion before changing the course.

GRAPHICS ERROR 207: Attempt to 'MoveTo' while moving to a destination

You must stop motion before changing course.

GRAPHICS ERROR 208: Attempt to 'FollowPath' while moving to a destination.

You must stop motion before changing the destination.

GRAPHICS ERROR 209: Path array must contain at least 2 points

Make sure at least two points that have been defined in the array passed to **FollowPath**.

GRAPHICS ERROR 210: Attempt to perform 'RotateTo' while rotating to an angle

You must stop motion before changing the angle to rotate to.

GRAPHICS ERROR 211: Attempt to perform 'ScaleTo' while scaling to a factor

You must stop motion before changing the size you are scaling to.

GRAPHICS ERROR 212: Form must be attached to a visible window before updating or drawing it

You must add the menu bar or dialog box to a window before drawing it.

GRAPHICS ERROR 213: Attempt to add invalid object to dialog box

Make sure the dialog box is a container for the type of object you are trying to add to it.

GRAPHICS ERROR 214: Graph not attached to tree

You must attach the graph to a window or an image object before drawing or updating.

GRAPHICS ERROR 215: Attempt to replace a NIL object

Only non-nil objects may be replaced.

GRAPHICS ERROR 216: Attempt to re-associate object

You must disassociate an object before associating it again.

GRAPHICS ERROR 217: Object not associated

Object which you are trying to disassociate has not been associated.

GRAPHICS ERROR 218: Attempt to save to a NIL library

You must create an instance of the library by performing **NEW** before saving it.

GRAPHICS ERROR 219: Attempt to load from a NIL library

You must create an instance of the library by performing **NEW** before saving it.

GRAPHICS ERROR 220: Object not found in library

Make sure the object name you pass to **LoadFromLibrary** is in the library.

GRAPHICS ERROR 221: Library object of wrong type

Make sure the type of object asked to **LoadFromLibrary** matches the type of object created in SIMDRAW. Also make sure that you perform the **INHERITED ObjInit** for any graphic objects which have **ObjInit** overridden, otherwise the object type is not defined.

GRAPHICS ERROR 222: Image's tree not attached to window

An icon's parent or ancestor must be attached to a window before the icon is drawn or updated.

GRAPHICS ERROR 223: Image not attached to tree

An icon must be attached to a window or an image before drawing or updating it.

GRAPHICS ERROR 224: Image's window not drawn

An image must be attached to a window that is visible before drawing or updating the icon.

GRAPHICS ERROR 225: Low world boundary equal to high

Make sure the **SetWorld** call for the image does not set the low values equal to the high values.

GRAPHICS ERROR 226: Attempt to add non-icon to an Image

Only images and their derived types may be added to an image.

GRAPHICS ERROR 227: Attempt to add icon already in tree

You must remove the image before adding it again.

GRAPHICS ERROR 228: Attempt to add non-list box item to listbox

Only list box items may be added to a list box.

GRAPHICS ERROR 229: Attempt to add non-menu items to menu bar

Only menus may be added to a menu bar.

GRAPHICS ERROR 230: Attempt to add non-menu item to menu

Only menu items may be added to a menu.

GRAPHICS ERROR 231: Slice number not found

Pie slice numbers start at 1. Make sure you have created as many slices in SIMDRAW as you are trying to access from the program.

GRAPHICS ERROR 232: Data set number not found

Data set numbers start at 1. Make sure you have created as many data sets in SIMDRAW as you are trying to access from the program.

GRAPHICS ERROR 233: Attempt to add non-radio button object to radio box

Only radio buttons may be added to a radio box object.

GRAPHICS ERROR 234: Non-existent library file <file name>

You tried to read in a library file that does not exist. This error does not occur if the library is asked to **SetNoError(TRUE)**.

GRAPHICS ERROR 235: Bad version library file

You tried to read in a library file created by a version of the editor newer than the runtime library you are using.

GRAPHICS ERROR 236: Snap shot file "filename" could not be read

An attempt was made to load in a bitmap file that was either non-existant, read protected, or of the wrong format. To disable printing of this error, use the **SnapShotObj.SetNoError** method, or set the **IgnoreSSLoadErrors** variable in the module **GSnap** to **TRUE**. In this case, the **FileStatus** field will contain the status of the last "Read" attempt.

GRAPHICS ERROR 237: Attempt to add a control to a file dialog box.

The structure of the file, font, and message dialog boxes is defined by the toolkit. You are not allowed to add controls to these objects.

GRAPHICS ERROR 238: System font family <*family name*> size: <*pt_size*> weight: <*weight*> could not be found

A font matching the given font desciption could not be found on the server. The **TextObj.SetNoFontError** method can be used to disable printing of this error message. In this case the **SysFontStatus** field will contain the status of the last attempt to display the text. See appendix D.

GRAPHICS ERROR 239: Object to convert not a WindowObj or ImageObj

Only objects deririved from **ImageObj** can be converted to EPS PostScript. Dialogs, menus and palettes cannot be converted.

GRAPHICS ERROR 240: Pane number exceeds total number of status panes

When using the **ShowStatus** or **SetPaneWidth** methods of **WindowObj**, the given pane number must be 0 and the number of panes - 1.

GRAPHICS ERROR 241: Button face icon <*path_nam*> could not be loaded

An attempt was made to load in a bitmap file for a palette button that was either non-existant, read protected, or of the wrong format. To disable printing of this error, use the **PaletteButtonObj.SetNoError** method, or set the **IgnoreSSLoadErrors** variable in the module **GSnap** to **TRUE**. In this case, the **FileStatus** field will contain the status of the last "Read" attempt.

Index

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z