Table of Contents

PREFACE.......................................................................................................................... 8
1.  OVERVIEW.................................................................................................................... 11
2.  2D GRAPHICS.............................................................................................................. 13
    2.1 GETTING STARTED................................................................................................. 13
5. 3D GRAPHICS

5.1.1 A Typical 3d Graphics Program

5.1.2 Example code for a typical 3d graphics program

5.1.3 Class hierarchy

5.2. The “SCENE-GRAPH”

5.2.1 The “3dworld”

5.2.2 Setting the location of a node

5.2.3 Setting the orientation of a node

5.2.4 Shifting the position of a node

5.2.5 Rotating a node

5.2.4 Scaling a node

5.2.5 Complete Scene-graph example

5.3.1 Camera location and orientation

5.3.2 Setting up the viewing plane

5.3.3 Adding a 3dcamera to the scene-graph

5.3.4 Viewports and multiple cameras

5.3.5 Tracking a moving object with the 3dcamera

5.4. LIGHTING UP A SCENE-GRAPH

5.4.1 Determining the “variety” of lighting

5.4.2 Ambient, Diffuse and Specular light

5.4.3 Setting up the lighting using the 3dlight

5.5 LOADING GRAPHICS FILES VIA MODELS

5.5.1 Reading in the 3dmodel from a file

5.5.2 Linking a 3dmodel instance to a 3dnode instance

5.5.3 Copying the scene-graph of a 3dmodel

5.5.4 Locating named sub-components

5.5.5 Deriving from sub-components

5.5.6 Misc. 3dmodel options

5.6 GEOMETRY

5.6.1 Using the 3dgraphic class

5.6.2 Surface geometry

5.6.4 Surface appearance

5.6.5 Surface texture mapping

5.6.6 Drawing points and lines

5.6.7: Drawing points

5.6.8 Text

5.6.9 Updating the graphic

5.6.8 Classes with retained geometry

5.7 SURFACES, TEXTURES AND MATERIALS

5.7.1 Setting the color and shininess of a surface

5.7.2 Texture mapping and raster images

5.7.3 Front and back side visibility
6. GRAPHICAL OBJECT REFERENCE ................................................................. 184

6.1 GUI.M OBJECTS....................................................................................................... 184

5.9 ANIMATION AND SIMULATION ............................................................................... 175

5.9.1 Frame Based Animation ................................................................................. 175
5.9.2 Time scaling........................................................................................................ 175
5.9.3 Automatic motion and the motion_set ............................................................ 176
5.9.4 Moving Car Example ..................................................................................... 177
5.9.5 Achieving customized motion......................................................................... 178

5.8 USER INPUT............................................................................................................ 165

5.8.1 Mouse, keyboard and window frame interaction........................................... 165
5.8.2 Mouse events..................................................................................................... 167
5.8.3 Keyboard input................................................................................................. 170
5.8.4 Selecting a node in the scene-graph ............................................................... 171

5.7.4 Using instances of 3dmaterial ........................................................................ 162
5.8.2 Mouse events..................................................................................................... 167
Value ....................................................................................................................... 175
Value ....................................................................................................................... 176
Value ....................................................................................................................... 177
Value ....................................................................................................................... 178
Value ....................................................................................................................... 179
Value ....................................................................................................................... 180

5.8.5 Achieving customized motion......................................................................... 181
5.8.6 Selecting a node in the scene-graph ............................................................... 182

5.7.4 Using instances of 3dmaterial ........................................................................ 162
5.8 USER INPUT............................................................................................................ 165
view (view → guiitem) .................................................................................................................. 206
window (window guiitem) .............................................................................................................. 207
windowevent (windowevent → guievent) .................................................................................... 210
6.2 3D.M OBJECTS ......................................................................................................................... 211
  3dcamera (3dcamera → 3dnode) .................................................................................................... 211
  3devent (3devent → gui.m:guiitem) ............................................................................................ 213
  3dfaces (3dfaces → 3dgraphic → 3dnode) .................................................................................. 214
  3dgraphic (3dgraphic → 3dnode) .................................................................................................. 219
  3dlight (3dlight → 3dnode) ............................................................................................................ 223
  3dlines (3dlines → 3dgraphic → 3dnode) .................................................................................... 224
  3dmaterial ..................................................................................................................................... 226
  3dmodel ....................................................................................................................................... 228
  3dnode ......................................................................................................................................... 231
  3dpoints (3dpoints → 3dgraphic → 3dnode) ................................................................................ 237
  3dtext (3dtext → 3dgraphic → 3dnode) .......................................................................................... 238
  3dwindow (3dwindow → gui.m:guiitem) ...................................................................................... 239
  3dworld ....................................................................................................................................... 242
6.3 3DSHAPES.M OBJECTS ................................................................................................................ 245
  3dbox (3dbox → 3dshape → 3dgraphic → 3dnode) ....................................................................... 245
  3dcone (3dcone → 3dshape → 3dgraphic → 3dnode) .................................................................... 245
  3dcylinder (3dcylinder → 3dshape → 3dgraphic → 3dnode) ....................................................... 245
  3dellipses (3dellipses → 3dshape → 3dgraphic → 3dnode) ............................................................ 245
  3drectangle (3drectangle → 3dshape → 3d.m:3dgraphic → 3d.m:3dnode) ................................. 246
  3dshape (3dshape → 3d.m:3dgraphic → 3d.m:3dnode) ................................................................. 247
  3dsphere (3dsphere → 3dshape → 3d.m:3dgraphic → 3d.m:3dnode) ........................................... 247
6.4 MISCELLANEOUS ROUTINES ................................................................................................. 248
  6.4.1 3d.m:handle.3devents.r ....................................................................................................... 248
  6.4.2 3d.m:compute_normal_vector ............................................................................................ 248
  6.4.3 3d.m:vector_normalize ....................................................................................................... 248
  6.4.4 3d.m:get_screen_size .......................................................................................................... 248
7. DEPLOYMENT GUIDE .................................................................................................................. 249
8. INDEX ......................................................................................................................................... 250
Table of Figures
Figure 1: Using the set_world and set_boundaries methods ........................................... 30
Figure 2: Using the set_world method to implement pan and zoom ............................... 31
Figure 3: Meters ................................................................................................................ 56
Figure 4: Clocks ............................................................................................................... 58
Figure 5: Bank model histogram....................................................................................... 61
Figure 6: Time trace plot................................................................................................. 65
Figure 7: A popup menu ................................................................................................... 87
Figure 8: MessageBox object............................................................................................ 92
Figure 9: Example 3d graphics program showing a 3d model of a 1963 ford ................. 101
Figure 10: 3d.m classes ................................................................................................... 102
Figure 11: 3dshapes.m classes ........................................................................................ 102
Figure 12: A typical scene graph for a tank and a battlefield ......................................... 103
Figure 13: Forward and Up vectors. (In this case the positive X axis would point away from the viewer) ............................................................................................................. 105
Figure 14: Calling rotate_z followed by move ............................................................... 107
Figure 15: Example 30 – A tank maneuvering through traffic cones ......................... 110
Figure 16: Camera orientation for Example 30 .............................................................. 112
Figure 17: Using the set_perspective method .............................................................. 113
Figure 18: Using the set_orthographic method ............................................................. 114
Figure 19: Using the set_viewport method ................................................................ 115
Figure 20: Example 31, tank program updated to include two viewports and tracking 119
Figure 21: The spot_cutoff attribute ............................................................................ 120
Figure 22: Three positional lights used in the parking lot demo ..................................... 121
Figure 23: Objects on the left have more specular color and appear shiny .................... 123
Figure 24: Using a spot light to track the tank ............................................................... 125
Figure 25: Three 3dnode instances using the same model of a jet .................................. 127
Figure 26: Calling the load method instead of assigning the model attribute ............... 128
Figure 27: Normal vectors .............................................................................................. 136
Figure 28: _triangles format .......................................................................................... 136
Figure 29: _triangle_strip format ................................................................................. 137
Figure 30: _triangle_fan format .................................................................................... 137
Figure 31: _quads format ............................................................................................... 138
Figure 32: _quad_strip format ...................................................................................... 138
Figure 33: _polygon format .......................................................................................... 139
Figure 34: A torus composed of rings made of quad strips .......................................... 139
Figure 35: Example 33 -- A torus drawn using the _quad_strip format ....................... 142
Figure 36: Example 34 -- A texture mapped torus ....................................................... 146
Figure 37: Using the _line_loop format to mark planetary orbits ................................ 147
Figure 38: The _lines format ......................................................................................... 147
Figure 39: The _line_strip format .................................................................................. 148
Figure 40: The _line_loop format .................................................................................. 148
Figure 41: Line widths and constants passed to the set_pen_pattern method ......... 149
Figure 42: Example 35 output .................................................................................... 151
Figure 43: The text fonts ............................................................................................... 153
This document contains information on CACI's new SIMSCRIPT III, Modular Object-Oriented Simulation Language, designed as a superset of the widely used SIMSCRIPT II.5 system for building high-fidelity simulation models. It focuses on the description of the SIMSCRIPT III Graphics.

CACI publishes a series of manuals that describe the SIMSCRIPT III Programming Language, SIMSCRIPT III Object-Oriented 2-D and 3-D Graphics and SIMSCRIPT III development environment SimStudio. All documentation is available on SIMSCRIPT WEB site http://www.simscript.com/products/simscript_manuals.html

• SIMSCRIPT III Graphics Manual —this manual – is a detailed description of the presentation graphics and animation environment for SIMSCRIPT III.
• **SIMSCRIPT III User’s Manual** – A detailed description of the SIMSCRIPT III development environment: usage of SIMSCRIPT III Compiler and the symbolic debugger from the SIMSCRIPT development studio, Simstudio, and from the Command-line interface.

• **SIMSCRIPT III Programming Manual** – A short description of the programming language and a set of programming examples.


Since SIMSCRIPT III is a superset of SIMSCRIPT II.5, a series of manuals and text books for SIMSCRIPT II.5 language, Simulation Graphics, Development environment, Data Base connectivity, Combined Discrete-Continuous Simulation, can be used for additional information:

• **SIMSCRIPT II.5 Simulation Graphics User’s Manual** — A detailed description of the presentation graphics and animation environment for SIMSCRIPT II.5

• **SIMSCRIPT II.5 Data Base Connectivity (SDBC) User’s Manual** — A description of the SIMSCRIPT II.5 API for Data Base connectivity using ODBC

• **SIMSCRIPT II.5 Operating System Interface** — A description of the SIMSCRIPT II.5 APIs for Operating System Services

• **Introduction to Combined Discrete-Continuous Simulation using SIMSCRIPT II.5** — A description of SIMSCRIPT II.5 unique capability for modeling combined discrete-continuous simulations.

• **SIMSCRIPT II.5 Programming Language** — A description of the programming techniques used in SIMSCRIPT II.5.

• **SIMSCRIPT II.5 Reference Handbook** — A complete description of the SIMSCRIPT II.5 programming language, without graphics constructs.

• **Introduction to Simulation using SIMSCRIPT II.5** — A book: An introduction to simulation with several simple SIMSCRIPT II.5 examples.

• **Building Simulation Models with SIMSCRIPT II.5** — A book: An introduction to building simulation models with SIMSCRIPT II.5 with examples.

The SIMSCRIPT language and its implementations are proprietary software products of the CACI Products Company. Distribution, maintenance, and documentation of the SIMSCRIPT language and compilers are available exclusively from CACI.
Free Trial Offer

SIMSCRIPT III is available on a free trial basis. We provide everything needed for a complete evaluation on your computer. There is no risk to you.

Training Courses

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

San Diego, California
Washington, D.C.

On-site instruction is available. Contact CACI for details.

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

CACI Products Company
1455 Frazee Road, suite 700
San Diego, California 92108
Telephone: (619) 881-5806
www.simscript.com
1. Overview

The goal of a simulation is to increase the understanding of the operation of a complex system. Unfortunately, the results of simulation studies are often presented only as pages of numbers, which fail to communicate the understanding gained. The complexity of the system and the simulation can make it difficult for users and decision makers to fully appreciate the interactions between elements of the system. In SIMSCRIPT, animated graphics can be used to clearly show the operation of the simulated system and graphic results are easily evaluated. System operation is better understood, and decision makers have more confidence in the simulation results. Graphical representation also facilitates debugging. Coding, data and modeling errors are apparent, thus avoiding the need for tedious error tracking. SIMSCRIPT III has a wide range of applications, and is prepared with built in language features allowing the programmer to represent entities ranging from aircraft on runways to messages in a communications network.

SIMSCRIPT III supports both 2d and 3d graphics for use in an object oriented framework. Simulated objects can hold a graphical representation and be moved through a background in a realistic manner. Various file formats such as .jpg and .3ds are supported, or you can design your own graphical objects using the SIMSCRIPT Studio Icon editor.

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

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

The SIMSCRIPT Studio development environment provides a point and click interface for creating objects falling into all three of the above categories of icons, graphs and forms. SIMSCRIPT provides both language and runtime support allowing these objects to be displayed in your program. The objects are contained in a graphics library file called graphics.sg2 shown as a component in your SIMSCRIPT Studio project tree. The names
of all graphical objects can be seen by clicking on the (+) next to "graphics.sg2". Any existing object can be shown in a separate editor by double clicking on its name. You can add new objects to the library by right-clicking on "graphics.sg2" then selecting "new" from the popup menu.

For SIMSCRIPT, graphics are split into 2 general categories – 2d and 3d graphics. Objects that relate to the "2d" category include 2-dimentional images, charts/plots and meters, and also the dialog boxes, menus and palettes. These objects are all defined in the "gui.m" subsystem (located in the "defs" folder under the SIMSCRIPT "home" directory. These objects can also coexist in the same window.

3d graphics form the other category. These objects are defined in the "3d.m" and "3dshapes.m" subsystems. 3d graphical objects currently must reside in a special 3d window which can only contain 3d objects. However both 2d and 3d windows can be used simultaneously in the same application.

In this book, Chapter 2 will cover 2-dimentional graphics or "icons", how to these icons using the SimStudio Icon editor, how to display the icons through a "view" and how to create, show and utilize windows. In Chapter 3, 2d meters and plots are described. Chapter 4 deals with "forms". All user interface items devoted to interactive getting input from the user are covered such as dialog boxes, palettes, menubars, tables, etc. Chapter 5 will cover all 3d graphics. Chapter 6 is a reference chapter that contains a section on each individual object found in the gui.m, 3d.m, or 3dshapes.m subsystems. Note that Chapters 2,3, and 4 relate only to objects found in the gui.m subsystem while Chapter 5 describes objects found in both 3d.m and 3dshapes.m subsystems.
2. 2d Graphics

SIMSCRIPT III Graphics is implemented as a set of classes supplied in a GUI.M subsystem/module, which is part of every SIMSCRIPT III distribution.

The GUI.M module contains classes that can be used to develop a graphical user interface for a SIMSCRIPT III model. Included is support for windows, dialog boxes, menu bars, palettes, graphics, icons and presentation graphs.

GUI.M is an interface to SIMGRAPHICS III and supports the same collection of features offered by the display entities and procedural interface provided in SIMSCRIPT II.5. GUI.M supports the loading of “.sg2” files containing icons, graphs and forms created by the SimStudio. Existing dialog boxes, icons, palettes etc. created for a SIMSCRIPT II.5 application are fully compatible with GUI.M. This allows an application to be fully object-oriented and to provide all the capabilities supported in SIMSCRIPT graphics.

The Simescript III 3d graphics package is an object-oriented wrapper around the OpenGL toolkit. The public preamble 3d.m.sim contains the basic class descriptions that allow the programmer to create windows containing 3d graphical scenes. “Camera” and “light” objects can be created to implement realistic scenes in an object-oriented fashion.

2.1 Getting Started

Using the objects provided in GUI.M, the application constructs a hierarchy of graphical objects. This hierarchy constructed by filing instances of GUI.M objects into sets owned by other instances of GUI.M objects. Generally speaking, GUI.M classes are used in the following manner:

1. Create an instance of an object found in GUI.M, or a suitably derived object.
2. Assign attributes to the instance.
3. File the instance in a set owned by another object that is to contain it.
4. Call the object methods to view or perform other tasks.

The Window object acts as a container for dialog boxes, palettes, a menu bar, as well as graphics that appear within its canvas. More specifically, a window owns a form_set containing objects derived from the Form and a view_set containing View objects. Form objects include dialog boxes, menu bars and palettes that are attached to a window. View objects represent a coordinate mapped region of the canvas of a window and act as containers for graphs and icons.

The GUI.M module supports loading objects derived from Icon, Graph or Form classes from the “graphics.sg2” file created by one of the SimStudio graphical editors. The
appearance attribute declared by these classes can be assigned before the object is displayed, allowing the information saved by SimStudio to define the characteristics of the Icon, Graph or Form.
A simple GUI application might perform the following steps:

1. Create an instance of a Window, assign its position, title attributes. Then call the `display` method to show the window.
2. Create a View, assign its world coordinate system attributes, file it in the `view_set` owned by the window.
3. Create an Icon, assign its `appearance`, file it in the `graphic_set` owned by the `view`. Then call the `display` method to show the icon.

The following program (Example1.sim) brings up a window to show an icon created by the SimStudio icon editor.

```
''Example1.sim
preamble including the gui.m subsystems
end
main
define my_window as Window reference variable
define my_view as a View reference variable
define my_icon as an icon reference variable

''create the window and show it on the screen
create my_window
let title(my_window) = "Example 1: Showing a simple icon"
call display(my_window)

''create a view to hold the icons, graphics, and graphs
create my_view
file this my_view in view_set(my_window)

''create an icon, load its appearance from SimStudio
create my_icon
let appearance(my_icon) = Templates'find("simple icon")
file this my_icon in graphic_set(my_view)

''show the icon in the canvas
call display(my_icon)

''wait for user to close the window
while 1=1
   call handle.events.r(1)
end
```
2.2 Objects Found in GUI.M

All objects that can be displayed on the computer screen are derived from a guiItem object. The diagram below shows the hierarchy. Derived objects are shown to the right of their base classes.

The Form, GuiItem and Graph classes are to be used as base classes and are not meant to be used in a create statement. The following table lists all classes defined in GUI.M. These classes will be described in greater detail later.
<table>
<thead>
<tr>
<th>Name</th>
<th>Create</th>
<th>Super-classes</th>
<th>Belongs In set</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock</td>
<td>Yes</td>
<td>graph</td>
<td>graphic_set</td>
<td>A graph appearing inside a window that shows simulation time.</td>
</tr>
<tr>
<td>color</td>
<td>No</td>
<td></td>
<td></td>
<td>A utility class for creating and decomposing a color.</td>
</tr>
<tr>
<td>dialogbox</td>
<td>Yes</td>
<td>form</td>
<td>form_set</td>
<td>Contains various data fields that can be viewed and adjusted by the user.</td>
</tr>
<tr>
<td>field</td>
<td>Yes</td>
<td>guiItem</td>
<td>field_set</td>
<td>A data field contained in a dialog box, menu bar or palette.</td>
</tr>
<tr>
<td>fillStyle</td>
<td>Yes</td>
<td></td>
<td></td>
<td>Defines patterns for drawing polygons, circles and pies. Used by the Graphic class.</td>
</tr>
<tr>
<td>form</td>
<td>No</td>
<td>guiItem</td>
<td>form_set</td>
<td>Base class for dialog boxes, message box, menu bar, and palette. Appearance must be defined using SimStudio.</td>
</tr>
<tr>
<td>formevent</td>
<td>No</td>
<td></td>
<td></td>
<td>Passed to the action method during interaction with form. Its attributes identify which data field was clicked on or changed.</td>
</tr>
<tr>
<td>graph</td>
<td>No</td>
<td>graphic</td>
<td>graphic_set</td>
<td>Base class for all 2d charts, clock, dial, level meters, etc. Appearance must be defined in SimStudio.</td>
</tr>
<tr>
<td>graphic</td>
<td>Yes</td>
<td>guiItem</td>
<td>graphic_set</td>
<td>Base class for all objects that can appear in the window canvas. Can be used to draw polygons, circles, lines, etc.</td>
</tr>
<tr>
<td>graphicEvent</td>
<td>No</td>
<td></td>
<td></td>
<td>Passed to the action method when a graphic is clicked on with the mouse.</td>
</tr>
<tr>
<td>guiItem</td>
<td>No</td>
<td></td>
<td></td>
<td>Base class for all objects that can be displayed on screen.</td>
</tr>
<tr>
<td>icon</td>
<td>Yes</td>
<td>graphic</td>
<td>graphic_set</td>
<td>A movable icon created in the SimStudio icon editor. Can be given a velocity and connected to a Simulation.</td>
</tr>
<tr>
<td>linefont</td>
<td>Yes</td>
<td>textFont</td>
<td></td>
<td>Predefined fonts for drawing scalable (vector based) text. Used by the Graphic class.</td>
</tr>
<tr>
<td>linestyle</td>
<td>Yes</td>
<td></td>
<td></td>
<td>Defines dash styles and widths for drawing arcs and polylines. Used by the Graphic class.</td>
</tr>
<tr>
<td>markstyle</td>
<td>Yes</td>
<td></td>
<td></td>
<td>Defines types of markers that can be drawn. Used by the Graphic class.</td>
</tr>
<tr>
<td>messagebox</td>
<td>Yes</td>
<td>form</td>
<td>form_set</td>
<td>Simple dialog box that can be defined at runtime or by SimStudio. Includes predefined response buttons.</td>
</tr>
<tr>
<td>meter</td>
<td>Yes</td>
<td>graphic</td>
<td>graphic_set</td>
<td>Graph that shows a single numerical value.</td>
</tr>
<tr>
<td>palette</td>
<td>Yes</td>
<td>form</td>
<td>form_set</td>
<td>A palette or control panel attached to an edge of the window containing rows of buttons. The buttons can be toggled or dragged onto the canvas. Derived objects can override the action method to receive immediate notification of user input.</td>
</tr>
<tr>
<td>piechart</td>
<td>Yes</td>
<td>graphic</td>
<td>graphic_set</td>
<td>A pie shaped graph representing a 1-dim array.</td>
</tr>
<tr>
<td>Name</td>
<td>Yes/No</td>
<td>Category</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>plot</td>
<td>Yes</td>
<td>graphic</td>
<td>A 2-d chart that can display a histogram or trace plot.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>guiitem</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>graphic_set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>popupmenu</td>
<td>Yes</td>
<td>form</td>
<td>A context menu usually displayed when a user right-clicks in the canvas of a Window object</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>guiitem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>systemfont</td>
<td>Yes</td>
<td>textfont</td>
<td>Contains font name, italic, point_size for drawing non-scalable “raster” text. Used by the Graphic class.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>template</td>
<td>No</td>
<td></td>
<td>An instance corresponds to an Icon, Graph or Form appearance as defined in SimStudio. The instance can be returned using the Templates’ find method.</td>
<td></td>
</tr>
<tr>
<td>templates</td>
<td>No</td>
<td></td>
<td>Utility class for reading files created by SimStudio. Creates a template for each graphical item saved in SimStudio.</td>
<td></td>
</tr>
<tr>
<td>textfont</td>
<td>No</td>
<td></td>
<td>Base class for both LineFont and SystemFont classes.</td>
<td></td>
</tr>
<tr>
<td>view</td>
<td>Yes</td>
<td>graphic</td>
<td>Defines a coordinate system as well as a holding area for Graphic objects. It can occupy a rectangular region in the canvas.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>guiitem</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>view_set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>window</td>
<td>Yes</td>
<td>guiitem</td>
<td>A resizable, scrollable window containing graphics and forms.</td>
<td></td>
</tr>
<tr>
<td>windowEvent</td>
<td>No</td>
<td></td>
<td>Passed to the action method of a Window when user does a click, drag, or adjusts the scrollbar.</td>
<td></td>
</tr>
</tbody>
</table>
2.3 Using Items Created in SimStudio

In SIMSCRIPT III icons, graphs, and forms are designed in one of the SimStudio graphical editors. Definitions created by the graphics editors are saved in a file with the extension “.sg2”. (When a new project is created, a default file named “graphics.sg2” is created for the project). GUI.M provides a way for the application to load these saved definitions and associate them with an existing object. An instance of a Template object will represent the saved definition of a icon, graph, or form. A template instance can be assigned to the appearance attribute of any object derived from Icon, Graph or Form.

Template objects are not created in the application code, but instead are returned by the find method of a utility class called Templates. This class acts as a manager of templates that have been read into the application from the “.sg2” file saved by SimStudio. During initialization, the “graphics.sg2” file will be loaded automatically. At this time all icon, graph and form definitions from this file will be stored in memory. The Templates'find method can then be called to return a template. This method takes as its only argument the name given to the item in SimStudio.

The Templates'read_sg2 method allows the application to read any “.sg2” file saved by SimStudio. In the following example, three identical icons are shown in a window. A template for these icons is obtained by the Templates class. The icons are saved in the file “Example2.sg2” which is read by the Templates class.

```
''Example2.sim
preamble including the gui.m subsystems
end

main
define window as Window reference variable
define view as a View reference variable
define dialog as a Dialogbox reference variable
define icon1, icon2, icon3 as Icon reference variables
define car_template, dialog_template as Template reference variables

tell the template manager to load everything in "Example2.sg2"
call Templates'read_sg2("Example2.sg2")

get the templates to be used from the template manager
let car_template = Templates'find("car icon")
let dialog_template = Templates'find("quit dialog")

create the window and show it on the screen
create window
define title(window) = "Example 2: Interfacing with SimStudio"
call display(window)

create a view to hold the icons
create view
define this view in view_set(window)

create 3 icons
create icon1, icon2, icon3
```
''set the template of each icon
let appearance(icon1) = car_template
let appearance(icon2) = car_template
let appearance(icon3) = car_template

file this icon1 in graphic_set(view)
file this icon2 in graphic_set(view)
file this icon3 in graphic_set(view)

''show the icons in the canvas
call display_at(icon1)(5000.0, 15000.0)
call display_at(icon2)(15000.0, 15000.0)
call display_at(icon3)(25000.0, 15000.0)

''show a simple dialog box
create dialog
let appearance(dialog) = dialog_template
file this dialog in form_set(window)

''wait for user to hit any button in the dialog
call accept_input(dialog)
end
2.4. Windows

The Window class in GUI.M allows the application to create independent windows, each having optional scroll bars, an optional status bar and a title. Subclassing a window and overriding the action method allows the application to be notified immediately of user interaction with the scrollbars, or of mouse input events.

Objects derived from GuiItem must either be attached to the window, or attached to another object that is itself attached to a window. An object can be attached by filing it into one of the sets owned by a Window. Objects derived from Form (dialog boxes, menu bar, palette) must be filed into the form_set. Another set owned by window called the view_set contains View objects (which in turn contains objects derived from Graphic such as the Icon and Graph).

Before being displayed assignment of the window’s properties should be made. Note that even though some properties might be specifically defined as “methods”, the expression referring to the property can be used on the left of the assignment operator (SIMSCRIPT III allows this). The display method should be called to make the window visible.

2.4.1 Size and Position

The position_xlo, position_ylo, position_xhi, and position_yhi properties will specify the geometry of the window. Coordinate values applied to these attributes should range from 0 to 32767. The point (0,0) defines the lower left corner of the computer screen, and (32767,32767) marks the upper right corner. Window size and position specifications include title bar, border and menu bar, (a window whose position_yhi is 16383 will NOT overlap another window whose position_ylo is 16383).

2.4.2 Canvas

The “canvas” of a window is the rectangular area inside the frame. This represents the drawing area for the application. Instances of View objects can appear in the canvas by filing them into the view_set owned by the Window object. Each View encompasses a rectangular section of the canvas and in turn contains Graphic objects (animated icons, backgrounds, and presentations graphs, etc.). See Chapter 2.5 for more information on using the View.
2.4.2.1 Canvas Coordinates

Each View defines a coordinate system for the objects it contains. The View objects themselves obey a fixed “canvas” coordinate system. The canvas coordinate system is always square, and (0,0) is mapped to the lower left corner of the canvas, while (32768,32768) maps to the upper right corner (if the window is square). The window, however, may be sized to be non-square by the user. There are three choices as to how to map the square canvas coordinates to a rectangular window canvas. The crop_mode property can be assigned to one of the following values:

_crop_none: Canvas coordinates will occupy the largest centered square within the canvas. All of coordinate system will be visible, but there may be gaps at the ends depending on how tall or wide the window is made.

_crop_top: The maximum “x” coordinate (32768) is fixed to the right border. The top portion of the canvas coordinates will not be visible if the window is wider than it is tall.

_crop_bottom: The maximum “y” coordinate (32768) is fixed to the top border. The right portion of the canvas coordinates will not be visible if the window is taller than it is wide.

The get_viewable_area method can be called to discover the visible canvas coordinate space in the window. Canvas coordinates are generally the default coordinate system used by Graphic objects. In other words, of no other coordinate system is explicitly described, canvas coordinates are used.

2.4.2.2 Background Color

The default background color for the window canvas is black. The color property of the window can be assigned to change the background color. A suitable color value can be returned from Color’RGB method. For example, to set the background color to red:

```
let color(my_window) = Color’rgb(1.0, 0.0, 0.0)
```

2.4.2.3 Printing the Canvas

The print method can be called to send the entire graphical content of the canvas to a printer. Pass “1” to the method to show a dialog box which allows the user to set printing options.
2.4.3 Title and Status Bar

A title bar is included at the top of every window. The title property can be assigned to change the text displayed in the title bar. A status bar can be shown at the bottom of the window frame. The status bar is composed of several panes, each containing a short text message that can be set at runtime.

The status_pane_count property must be set to a non-zero integer to enable the status bar. Each pane in the status bar should be assigned a width (in characters) using the status_pane_width property. The width of the first pane is determined automatically based on the width of the window. Therefore, the width specification for the first status pane (i.e. status_pane_width(window)(1)) is always ignored. The status_pane_text property sets the text in an individual pane.

2.4.4 Scroll bars

Both horizontal and vertical scroll bars can be displayed in a window. Scroll bars can provide a natural mechanism for panning across a scene too large to fit inside the boundaries of your window. This is common after zooming into a rectangular section of the canvas. To create the scrollable window, set the scrollable_h and/or scrollable_v properties to “1 before displaying the window.

You can set the width of a scroll bar thumb before or after the window has been displayed. This is accomplished via the thumb_size_h and thumb_size_v properties. These values should range from 0.0 to 1.0. This represents the percentage of the total scroll bar area you wish the thumb to occupy. If the scroll bars are used for pan and zoom, the thumb size should be equal to the ratio of viewable area to total area.

The thumb_pos_h and thumb_pos_v attributes hold the distance between the thumb and the left (or top) of the scroll bar. Possible values for thumb_pos_h and thumb_pos_v are in the range [0.0 .. 1.0- thumb_size_h] and [0.0 .. 1.0- thumb_size_v], respectfully.
2.4.5 Handling User Input

When a user resizes, moves, scrolls, or closes a window, it may be necessary for the program to take some sort of action. For example, if the user moves a scroll bar, you may want to "pan" the contents of the window. Applications needing to implement asynchronous notification should subclass the Window object and override the action method. This method is called automatically whenever the user moves or clicks in the window. Code can then be provided in this method to handle the event.

The action method takes a WindowEvent as a parameter. The id attribute of the WindowEvent indicates the type of event. The id will contain one of the constants defined in the WindowEvent class. The WindowEvent object holds additional event specific data that can be utilized by the action method. The following table lists the id constants, a description of the particular action, and which WindowEvent attributes can be used by the application.
<table>
<thead>
<tr>
<th>EVENT ID</th>
<th>CAUSE OF EVENT</th>
<th>EVENT ATTRIBUTES USED</th>
</tr>
</thead>
<tbody>
<tr>
<td>_activate</td>
<td>Window has been brought to the front of all other windows.</td>
<td>key_code, key_literal</td>
</tr>
<tr>
<td>_close</td>
<td>Window has been closed. (User clicked on the “X” in the upper corner.)</td>
<td>key_code categorizes the key as alphanumeric or function ( _literal_key, _f1_key, _f2_key,...)</td>
</tr>
<tr>
<td>_key_down</td>
<td>User has pressed down on key</td>
<td>key_code, key_literal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>_key_literal is the alpha value of the key. Only valid if key_code is _literal_key</td>
</tr>
<tr>
<td>_key_up</td>
<td>User has let up on a key</td>
<td>key_code, key_literal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_mouse_down</td>
<td>User has clicked down inside the window canvas with any button.</td>
<td>button_number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 =&gt; left mouse button</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2,3 =&gt; middle/right button</td>
</tr>
<tr>
<td></td>
<td></td>
<td>click_count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 =&gt; single click</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>x,y Location of click in CANVAS coordinates range [0..32767].</td>
</tr>
<tr>
<td>_mouse_up</td>
<td>User has clicked down inside the window canvas with any button. Also called after a double-click.</td>
<td>button_number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 =&gt; left mouse button</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2,3 =&gt; middle/right button</td>
</tr>
<tr>
<td></td>
<td></td>
<td>click_count</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 =&gt; single click</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 =&gt; double click</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x,y Location of click in CANVAS coordinates range [0..32767].</td>
</tr>
<tr>
<td>_mouse_move</td>
<td>User has moved the mouse inside the window canvas.</td>
<td>x,y Location of mouse in CANVAS coordinates range [0..32767].</td>
</tr>
<tr>
<td>_reposition</td>
<td>The window has been dragged to a new location.</td>
<td>x,y Location in screen coordinates. Range is [0..32767]</td>
</tr>
<tr>
<td>_resize</td>
<td>The window has been resized.</td>
<td>x,y Size in screen coordinates. Range is [0..32767]</td>
</tr>
<tr>
<td>_scroll_x</td>
<td>The horizontal scroll bar has been used.</td>
<td>x Location of horiz. Thumb. Range is [0..1]</td>
</tr>
<tr>
<td>_scroll_y</td>
<td>The vertical scroll bar has been used.</td>
<td>y Location of vert. Thumb. Range is [0..1]</td>
</tr>
</tbody>
</table>
2.4.6 Reading Mouse Input Synchronously

Input from a *Window* object can also be handled synchronously. The method `accept_input` blocks execution until the user clicks in the canvas of the window. An anchored rubber band cursor which tracks the pointer can be shown. `accept_input` yields both the (x,y) coordinates of the mouse click, and a pointer to the *View* that was clicked in. These coordinates are bounded by the coordinate system assigned to the yielded *View*.

`accept_input` parameters include the (x,y) location (in NDC coordinates) of the anchor point of the cursor as well as an integer representing the cursor style. Available cursors are shown in the table below.

<table>
<thead>
<tr>
<th>CONSTANT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>_cursor_none</td>
<td>Do not show a cursor.</td>
</tr>
<tr>
<td>_cursor_line</td>
<td>Show a rubber band line anchored at the given anchor point. The line is updated automatically as the mouse is moved.</td>
</tr>
<tr>
<td>_cursor_box</td>
<td>Show a rubber band box anchored at the given anchor point. The corner of the box is updated automatically as the mouse is moved.</td>
</tr>
<tr>
<td>_cursor_icon</td>
<td>Use this cursor style if a <em>Graphic</em> object should be moved with the mouse. The object should be provided as a parameter.</td>
</tr>
</tbody>
</table>

2.4.7 Using a Window

If the full features of a *Window* need to be employed in the application, the following steps should be taken into account:

1. In your preamble, subclass the *Window* object and override its action method.
2. In the *action* method, write code to check the *id* attribute of the given *WindowEvent* and respond accordingly.
3. In the program initialization, create your *Window* object and assign properties such as position, title etc.
4. Add instances of *View* object(s) to the view_set. Add *Graphic* object(s) to the graphic_set owned by the *View*.
5. Call the *display* method to show the window. Objects filed into the view_set will also be shown.

The following example shows a window with scroll bars and a status bar. In this example the action method is overridden, allowing the events and associated data to be reported through a dialog box.

```
Preamble including the gui.m subsystem

'Example3.sim
'This example displays a window containing a status bar, scroll bars
'and a background image. Mouse and scroll bar interaction is reported
'immediately via a small dialog box.

'subclass the Window and override the action method
```
begin class MyWindow
    every MyWindow is a Window,
    overrides the action
end

define my_dialog as a DialogBox reference variable
end

'Action will be called upon user interaction with the window controls
'and with the mouse.
'Use a dialog box to list the WindowEvent attributes as user interacts
'with the window.
method MyWindow::action(event)
    let value(find(my_dialog)("button_number")) = button_number(event)
    let value(find(my_dialog)("click_count")) = click_count(event)
    let value(find(my_dialog)("x")) = x(event)
    let value(find(my_dialog)("y")) = y(event)
    let value(find(my_dialog)("key_code")) = key_code(event)
    let value(find(my_dialog)("key_literal")) = key_literal(event)
    let value(find(my_dialog)("modifiers")) = modifiers(event)

    select case id(event)
        case WindowEvent::activate
            let string(find(my_dialog)("id")) = "_activate"
        case WindowEvent::close
            let string(find(my_dialog)("id")) = "_close"
        case WindowEvent::key_down
            let string(find(my_dialog)("id")) = "_key_down"
        case WindowEvent::key_up
            let string(find(my_dialog)("id")) = "_key_up"
        case WindowEvent::mouse_down
            let string(find(my_dialog)("id")) = "_mouse_down"
        case WindowEvent::mouse_up
            let string(find(my_dialog)("id")) = "_mouse_up"
        case WindowEvent::mouse_move
            let string(find(my_dialog)("id")) = "_mouse_move"
        case WindowEvent::reposition
            let string(find(my_dialog)("id")) = "_reposition"
        case WindowEvent::resize
            let string(find(my_dialog)("id")) = "_resize"
        case WindowEvent::scroll_x
            let string(find(my_dialog)("id")) = "_scroll_x"
        case WindowEvent::scroll_y
            let string(find(my_dialog)("id")) = "_scroll_y"
    endselect
    call display(my_dialog)
    return with 1
end
main
    define my_window as a MyWindow reference variable
    define my_view as a View reference variable
    define my_icon as an Icon reference variable

    create my_window

    ''for non-square window, crop the top portion
    let crop_mode(my_window) = Window'_crop_top

    ''Place the window on the right side of screen
    let position_xlo(my_window) = 8000
    let position_ylo(my_window) = 2000
    let position_xhi(my_window) = 32768
    let position_yhi(my_window) = 32768

    ''Make the window scrollable in both the horizontal and
    ''vertical directions
    let scrollable_h(my_window) = 1
    let scrollable_v(my_window) = 1

    ''Make the thumbs 10% the size of the scroll bar.
    ''Position the top/left of the thumb to the 50% position.
    let thumb_size_h(my_window) = 0.10
    let thumb_size_v(my_window) = 0.10
    let thumb_pos_h(my_window) = 0.5
    let thumb_pos_v(my_window) = 0.5

    ''Specify 3 status bar panes. Set the pane sizes to hold
    ''text strings 10, and 15 characters respectfully.
    ''(The width of pane 1 is determined automatically)
    let status_pane_count(my_window) = 3
    let status_pane_width(my_window)(2) = 10
    let status_pane_width(my_window)(3) = 15
    let status_pane_text(my_window)(1) = "Pane 1 text"
    let status_pane_text(my_window)(2) = "Pane 2 text"
    let status_pane_text(my_window)(3) = "Pane 3 text"

    ''set the text displayed on the title bar
    let title(my_window) = "Example 3: A full featured window"

    ''create a view to appear in the window
    create my_view
    file this my_view in view_set(my_window)

    ''create an icon to appear in the view
    create my_icon
    let appearance(my_icon) = Templates'find("background")
    file this my_icon in graphic_set(my_view)

    ''display window and contents of canvas
    call display(my_window)

    ''dialogs are explained later
    create my_dialog
    let appearance(my_dialog) = Templates'find("example3 dialog")
    file this my_dialog in form_set(my_window)
    call display(my_dialog)

    while visible(my_dialog) <> 0
        call handle.events.r(1)
    end
2.5. View Object

The purpose of a View object is to define both a coordinate system and a bounding rectangle for the Graphic objects it contains. Each instance of a View object occupies a rectangular sub-region of the canvas. This feature allows multiple views to share the same canvas. A view should be “added” to a window canvas by filing it into the view_set owned by the window. Since the View object is derived from GuiItem, the display and erase methods can be used to show or hide all objects in the view_set.

2.5.1 View boundaries

The region occupied by the view is defined by calling the set_boundaries method. Arguments to set_boundaries specify a rectangle in the canvas and are given in window canvas coordinates (range: 0 to 32768).

\[
\text{call set_boundaries(view)(xlo, xhi, ylo, yhi)}
\]

The default boundaries for a view will encompass the entire window canvas (i.e. xlo=0, xhi=32768, ylo=0, yhi=32768). If the view should take up the entire window canvas, then it is not necessary to call set_boundaries.

2.5.2 Coordinate System for Graphic Objects

Each view also defines a coordinate system, of which applies to the Graphic objects filed into its graphics_set. All Graphic objects contained by the view will be positioned and scaled in size relative to this world coordinate system. The set_world method is called to provide the range of coordinates to the View object.

\[
\text{call set_world(view)(xmin, xmax, ymin, ymax)}
\]

There are no restrictions on the magnitude of these coordinates. However, if using an Icon object loaded from the SimStudio icon editor, the world coordinate system defined by the view should match the “SETWORLD.R parameters” found in the “Icon Properties” dialog box.
2.5.3 Overlapping Views

If two overlapping View objects occupy the same canvas, the stacking order is always defined by the ordering of the views in the view_set owned by the window. View objects filed last in the set will appear on top. All Graphic objects filed into a view obey the stacking order defined by the view, in other words objects contained by different views will not be shown interleaved.

2.5.4 Pan and Zoom

The set_world method can be used to implement panning and zooming for a background. A pan can be implemented by shifting the boundaries of the coordinate system to the left to pan left, and to the right to pan right. Specifying a smaller range of coordinate boundaries will appear to “zoom in”. In the following figure, a scene showing an original coordinate system of xmin=0, xmax=1000, ymin=0, ymax=1000 is zoomed by specifying a new coordinate system of xmin=300, xmax=700, ymin=500, ymax=900.
Figure 2: Using the set_world method to implement pan and zoom.

The following example shows a window with horizontal and vertical scrollbars. A View object containing a background is placed inside the canvas. When the user clicks in the canvas with the left mouse button, the view is “zoomed in” by a factor of 2. The click location is used as a center point for the zoom operation. Clicking with the right mouse button will zoom out all the way. Moving the scroll bars will “pan” the zoomed scene in the direction of the thumb movement. Both pan and zoom operations are implemented with the View’s set_world method.
This example displays a window containing scroll bars and a background image. Moving the scroll bar thumbs will pan, clicking with the left mouse button will zoom in, clicking with the right mouse button will zoom out all the way.

preamble including the gui.m subsystem
'subclass the Window and override the action method

begin class MyWindow
every MyWindow is a Window, and overrides the action
end

begin class MyView
every MyView is a View and has
  a zoom_factor,
a zoom method,
a pan method
define zoom as a method given
  1 double argument,  ''zoom factor
  2 double arguments  ''center of zoom area
define pan as a method given
  2 double arguments  ''x,y position of world
define zoom_factor as a double variable
end

'globals and constants
define scene_xlo=0, scene_ylo=0, scene_xhi=1000, scene_yhi=1000 as constants
define theWindow as a MyWindow reference variable
define theView as a MyView reference variable
end ''preamble
'"Action" will be called upon user interaction with scroll bars and
'the mouse.

method MyWindow'action(event)
define view_x, view_y as double variables

'if user clicks down on the left mouse button, zoom into that spot
'each click will zoom in by a factor of 2
if id(event) = WindowEvent'_mouse_click and button_down(event) <> 0 and
button_number(event) = 0
call get_view_xy(x(event), y(event)) yielding view_x, view_y, view
   call zoom(theView)(zoom_factor(theView) * 2, view_x, view_y)
always

'if user clicks down on the right mouse button, zoom out
if id(event) = WindowEvent'_mouse_down and button_down(event) <> 0 and
button_number(event) <> 0
   call zoom(theView)(1.0, (scene_xhi-scene_xlo) / 2.0, (scene_yhi-scene_ylo) / 2.0)
always

'handle scroll bar movement by user. This will cause the scene to pan
if id(event) = WindowEvent'_scroll_x or id(event) = WindowEvent'_scroll_y
   call pan(theView)(
      x(event) * (scene_xhi-scene_xlo) + scene_xlo,
      y(event) * (scene_yhi-scene_ylo) + scene_ylo)
always
return with 0
end

'"This method uses the "set_world" method of the View object to
'zoom in and out.

method MyView'zoom(factor, cx, cy)
define sizex, sizey as double variables

   let zoom_factor = factor
   let sizex = (scene_xhi - scene_xlo) / factor;
   let sizey = (scene_yhi - scene_ylo) / factor;

   'setting a new world will zoom in or out
   call set_world(cx - sizex / 2.0, cx + sizex / 2.0,
                cy - sizey / 2.0, cy + sizey / 2.0)
   'update the scroll bar thumb size and position
   let thumb_size_h(theWindow) = 1.0 / factor
   let thumb_size_v(theWindow) = 1.0 / factor
   let thumb_pos_h(theWindow) = ((world_xlo - scene_xlo) / (scene_xhi - scene_xlo))
   let thumb_pos_v(theWindow) = (scene_yhi - world_yhi) / (scene_yhi - scene_ylo)
   call display(theWindow)
end

'"This method uses the "set_world" method of the View object to
'pan left, right, up and down.

method MyView'pan(x, y)
   'to show the effect of panning, change the world boundaries
   call set_world(
      scene_xlo + x, scene_xlo + x + (world_xhi-world_xlo),
      scene_yhi - y - (world_yhi-world_ylo), scene_yhi - y)
end
define theIcon as an Icon reference variable

create theWindow

'for non-square window, crop the top portion
let crop_mode(theWindow) = Window'_crop_top

'Make the window scrollable in both the horizontal and
'vertical directions
let scrollable_h(theWindow) = 1
let scrollable_v(theWindow) = 1

'set the text displayed on the title bar
let title(theWindow) = "Example 4: Implementing Pan and Zoom"

'create a view to appear in the window
create theView
let zoom_factor(theView) = 1
call set_world(theView)(scene_xlo, scene_xhi, scene_ylo, scene_yhi)
file theView in view_set(theWindow)

'create an icon to appear in the view
create theIcon
let appearance(theIcon) = Templates'find("floor.icn")
file theIcon in graphic_set(theView)

'display window and contents of canvas
call display(theWindow)

while visible(theWindow) <> 0
call handle.events.r(1)
end
2.6. Graphics in a Window

The GUI.M module allows the application to not only animate icons created in SimStudio, but also custom defined graphics. The Graphic object provides the base class for all objects that can both appear in the canvas, and can be composed of a group of shape primitives. This includes the Icon and Graph objects. In order to draw into a window, the Window object must contain at least one View object (filed into its view_set). A Graphic object can then be made to appear inside a particular window by filing it into the graphic_set owned by one of the View objects belonging to the window. The Icon object inherits the capabilities of the Graphic, but can be constructed off-line in the SimStudio Icon editor. Icons can also be animated, in other words they can have a velocity and/or movement which is linked to the advancing simulation time.

2.6.1 Drawing Shape Primitives

Using the methods and attributes of the Graphic object, the application can draw groups of shape primitives and position them in the View. The shapes can also be rotated and scaled. Each shape has a unique color. “Style” objects can be created and are used in conjunction with draw methods to specify text font, line width, hatch styles, etc. The following table shows which shapes can be drawn:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>draw_arc</td>
<td>X, y, radius, start_angle, stop_angle, color, style</td>
<td>Draws a semi-circular arc. (x,y) marks the center, and the arc is drawn counter-clockwise from start_angle to stop_angle. Angles are given in radians and measured from the positive x axis. ALineStyle object reference should be passed for the ‘style’ argument.</td>
</tr>
<tr>
<td>draw_circle</td>
<td>x, y, radius, color, style</td>
<td>Draws a circle. (x,y) marks the center, and a radius should also be given. AFillStyle object reference should be passed for the ‘style’ argument.</td>
</tr>
<tr>
<td>draw_polygon</td>
<td>points, color, style</td>
<td>Adds an n-sided polygon to the icon. The first dimension of the points argument selects the X or Y coordinate. The second dimension contains the point number.</td>
</tr>
<tr>
<td>draw_polyline</td>
<td>points, color, style</td>
<td>Adds an n-point connected line segment to the graphic. First and last points are not connected. (Format of points array is identical to draw_polygon)</td>
</tr>
<tr>
<td>draw_polymark</td>
<td>points, color, style</td>
<td>Adds point markers to the Graphic object. (Format of points array is identical to draw_polygon)</td>
</tr>
<tr>
<td>draw_pie</td>
<td>x, y, radius, start_angle, stop_angle, color,</td>
<td>Draws a filled pie slice. (x,y) marks the center, and the pie is drawn counter-clockwise from start_angle to stop_angle. Angles are given in radians and measured from the positive x axis. AFillStyle object reference should be passed for the ‘style’ argument.</td>
</tr>
</tbody>
</table>
draw_rectangle

<table>
<thead>
<tr>
<th>style</th>
<th>Draws a rectangle given its lower left corner point (specified by x,y) and both its width and height.</th>
</tr>
</thead>
<tbody>
<tr>
<td>x,y, width, height, color, style</td>
<td></td>
</tr>
</tbody>
</table>

All calls to drawing methods made by a Graphic object instance should be bracketed by a call to the begin_drawing method and the end_drawing method. The begin_drawing method will eliminate all shapes previously drawn. The end_drawing method marks the end of a sequence of calls to draw_ methods.
### 2.6.2 Points, style, and color

A 2-dim array of doubles is used to represent the points defining the shape of the draw primitive. In this array, the first index defines which of the coordinate (x or y) the value refers to. The second index is the point number. For example, to create an array of points for a triangle formed by the verticies (-50,-50), (50,-50), (0,50) use the following code:

```
Reserve points(*,*) as 2 by 3
let points(1,1) = -50 let points(2,1) = -50  "south west corner"
let points(1,2) = 50  let points(2,2) = -50  "south east corner"
let points(1,3) = 0   let points(2,3) = 50   "north corner"
```

Style objects can be created, initialized and passed as an argument to the drawing methods. If a “0” is passed instead of the style object, default values are used. As a convenience, the built in global value for each style object is created automatically. (For example, the global variable name FillStyle can be passed directly to draw_polygon without being created by the application.)

<table>
<thead>
<tr>
<th>Style Object</th>
<th>Drawing methods</th>
<th>Attribute</th>
<th>Possible values</th>
</tr>
</thead>
<tbody>
<tr>
<td>FillStyle</td>
<td>draw_circle</td>
<td>pattern</td>
<td>_hollow, _solid, _narrow_diagonal, _medium_diagonal, _wide_diagonal, _narrow_crosshatch, _medium_crosshatch, _wide_crosshatch (default=_hollow)</td>
</tr>
<tr>
<td></td>
<td>draw_polygon</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>draw_pie</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LineStyle</td>
<td>draw_arc</td>
<td>pattern</td>
<td>_solid, _long_dash, _dotted, _dash_dotted, _medium_dash, _dash_dot_dotted, _short_dash, _alternate (default = _solid)</td>
</tr>
<tr>
<td></td>
<td>draw_polyline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MarkStyle</td>
<td>draw_polymark</td>
<td>type</td>
<td>_dot, _cross, _asterisk, _square, _x, _diamond (default=dot)</td>
</tr>
<tr>
<td></td>
<td>draw_pie</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>draw_text</td>
<td>font</td>
<td>_basic, _simple, _roman, _bold_roman, _italic, _script, _greek, _gothic, (default= _basic)</td>
</tr>
<tr>
<td>SystemFont</td>
<td>draw_text</td>
<td>bold</td>
<td>1 to indicate a bold face. 0 indicates plain.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>direction</td>
<td>_right=0, _up, _left, _down</td>
</tr>
<tr>
<td></td>
<td></td>
<td>family</td>
<td>Name of font face. i.e. &quot;Arial&quot;, &quot;Courier&quot;, etc... (default is system dependant)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>italic</td>
<td>1 to indicate an italic face, 0 for non-italics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>point_size</td>
<td>Size in points of the text. (default =12)</td>
</tr>
</tbody>
</table>

All drawing methods accept a color argument that can be used to paint the primitive. Colors are stored as integer values, but are generally specified as percentages of red, green, and blue components (range 0.0 to 1.0). The Color class provided class methods that can convert between this single integer value and the three RGB components. The
*Color*rgb method returns a color value given its RGB components. This value can be passed to a drawing method. The *Color*red, *Color*green and *Color*blue methods return the RGB component value of a color value.

For example, to draw a dark green solid triangle (using the points supplied above):

```plaintext
call begin_drawing(myGraphic)
let pattern(FillStyle) = FillStyle'_solid
call draw_polygon(myGraphic)(points, Color’rgb(0.0, 0.5, 0.0), FillStyle)
call end_drawing(myGraphic)
```
2.6.3 Location, Rotation, and Scale

Using the location, rotation, and scale attributes allows several different Graphic object instances to share the same “points” but still be shown in different locations with a different geometry. A Graphic object is positioned relative to the coordinate system assigned to the View object that it is attached to. The location_x and location_y properties contain the “x” and “y” offset to the center of the graphic (0,0) from the (0,0) coordinate in the View that the Graphic is filed into.

Setting the rotation property of a Graphic object will rotate all shapes drawn into the object counter-clockwise about the (0,0) point. The angle is specified in radians. The position of drawn text will be rotated, but the text itself will remain horizontal.

A scale factor can be applied to a Graphic object as well. All points in the drawn shapes are multiplied by this factor before the Graphic object is displayed. Setting the scale property to a value between 0 and 1 will therefore make the object smaller while values greater that 1 will make it big.

In the following example, a green triangle is drawn into a view with the coordinate system (xlo=0,ylo=0, xhi=100, yhi=100). The triangle is rotated and scaled continuously to make it appear to be “spinning away” from the viewer.
'Example5.sim
'This example shows a spinning triangle created by program code only.

preamble including the gui.m subsystem
end

main

define theWindow as a Window reference variable
define theGraphic as a Graphic reference variable
define theView as a View reference variable
define points as a 2-dim double variable

create theWindow
let title(theWindow) = "Example 5: Location, Rotation and Scale"

'create a view to appear in the window
create theView
call set_world(theView)(0, 100, 0, 100)
file theView in view_set(theWindow)

'create the points necessary to define the triangle
'design points so that (0,0) is the center of the triangle
Reserve points(*,*) as 2 by 3
let points(1,1) = -50    let points(2,1) = -50  "south west corner
let points(1,2) = 50     let points(2,2) = -50  "south east corner
let points(1,3) = 0      let points(2,3) = 50   "north corner

'create the graphic to hold the triangle
create theGraphic

'start the drawing, draw the polygon, then end the drawing
call begin_drawing(theGraphic)
let pattern(FillStyle) = FillStyle'_solid
call draw_polygon(theGraphic)(points, Color'rgb(0.0, 0.5, 0.0),
FillStyle)
call end_drawing(theGraphic)

'put the graphic in the middle of the View
let location_x(theGraphic) = 50.0
let location_y(theGraphic) = 50.0

file theGraphic in graphic_set(theView)

'display window and contents of canvas
call display(theWindow)

'loop until the user dismisses the window
while visible(theWindow) <> 0 and scale(theGraphic) > 0.01
do
'update rotation and scale with each loop
let rotation(theGraphic) = rotation(theGraphic) + 0.02
let scale(theGraphic) = scale(theGraphic) * 0.999

call display(theGraphic)  "redisplay after changes to geometry

call handle.events.r(0)  "handle any mouse events
loop
end
2.6.4 Responding to Clicks on a Graphic

Occasionally, the application will need to allow the user to click on a Graphic object while the simulation is running. The program can then for example display a dialog box containing information about the object, or perform other actions in response to the selection.

To allow users to click on a Graphic object, a new class must be derived from Graphic and the action method should be overridden, as is shown here:

```
preamble including the gui.m subsystem
begin class MyGraphic
    every MyGraphic is a Graphic and overrides the action
end

method MyGraphic'action(event)
    write as "MyGraphic was selected!", /
end
```

The event argument is a reference pointer to an object of class GraphicEvent which is derived from GuiEvent. The id attribute can take on the following values:

<table>
<thead>
<tr>
<th>GraphicEvent‘id</th>
<th>Description of event</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mouse_down</td>
<td>User clicked on object</td>
</tr>
<tr>
<td>_mouse_up</td>
<td>User released the mouse over the object</td>
</tr>
<tr>
<td>_mouse_enter</td>
<td>User moved mouse over the body of object</td>
</tr>
<tr>
<td>_mouse_leave</td>
<td>User moved mouse away from body of object.</td>
</tr>
</tbody>
</table>

The modifiers attribute of GraphicEvent can be used to determine if the user was holding down the “shift”, “alt” or “control” key while clicking. At the time the action method is called, the modifiers attribute will be the logical OR of the constants _shift_down, _ctrl_down and _alt_down. The click_count attribute of GraphicEvent will be “2” if the object was double-clicked on.

The following is a complete example showing the seven different shapes that can be created with the Graphic object. Graphic objects in this program can be clicked on.

```
'Example6.sim
'This example shows a all the different shapes that can be
'created using the Graphic object. Clicking on a shape with
'the mouse will display it.

preamble including the gui.m subsystem
begin class MyGraphic
    every MyGraphic is a Graphic and has
    a name, and
    overrides the action
    define name as a text variable
end
define theWindow as a Window reference variable
```
define theView as a View reference variable
define message_graphic as a Graphic reference variable
end

method MyGraphic'action(event)
  define act, mod as text variable
  ''use the "id" attribute to get the type of mouse activity
  select case id(event)
    case GraphicEvent'_mouse_down  let act = " was clicked on"
    case GraphicEvent'_mouse_up    let act = " was released over"
    case GraphicEvent'_mouse_enter let act = " was hovered into"
    case GraphicEvent'_mouse_leave let act = " was hovered out of"
    default  return with 0
  endselect

  let size(LineFont) = 400
  ''use the "modifiers" attribute to see if <shift>, <alt> or <ctrl>
  ''was used
  if and.f(modifiers(event), GraphicEvent'_shift_down) <> 0
    let mod = " with the shift key down"
  always
  if and.f(modifiers(event), GraphicEvent'_alt_down) <> 0
    let mod = mod + " with the alt key down"
  always
  if and.f(modifiers(event), GraphicEvent'_ctrl_down) <> 0
    let mod = mod + " with the ctrl key down"
  always

  ''create a little message that will show the name of the object
  call begin_drawing(message_graphic)
  call draw_text(message_graphic)(50, 950, Color'_white, LineFont, name + act + mod)
  call end_drawing(message_graphic)
  call display(message_graphic)
  return with 0
end
main
  define aGraphic as a MyGraphic reference variable
  define points as a 2-dim double variable

  create theWindow
  let title(theWindow) = "Example 6: Shapes that a Graphic object can draw"

  ''create a view to appear in the window
  create theView
  call set_world(theView)(0, 1000, 0, 1000)  ''world from 0 to 1000
  file theView in view_set(theWindow)

  ''create a message to appear when objects are clicked on
  create message_graphic
  file this message_graphic in graphic_set(theView)

  ''1) draw a square polygon.  First, define points, style
  Reserve points(*,*) as 2 by 4  ''define corner points
  let points(1,1) = 100    let points(2,1) = 800
  let points(1,2) = 200    let points(2,2) = 800
  let points(1,3) = 200    let points(2,3) = 900
  let points(1,4) = 100    let points(2,4) = 900

  ''now create a "MyGraphic" instance.
  create aGraphic
  file this aGraphic in graphic_set(theView)
  let name(aGraphic) = "blue polygon"
  call begin_drawing(aGraphic)
  let pattern(FillStyle) = FillStyle'_solid
  call draw_polygon(aGraphic)(points, Color'rgb(0,0,1.0), FillStyle)
  let pattern(FillStyle) = FillStyle'_hollow
  call draw_polygon(aGraphic)(points, Color'_cyan, FillStyle)
  call end_drawing(aGraphic)

  ''2) draw a polyline.  Use the same points as above
  create aGraphic
  file this aGraphic in graphic_set(theView)
  let name(aGraphic) = "red polyline"
  call begin_drawing(aGraphic)
  let pattern(LineStyle) = LineStyle'_long_dash
  call draw_polyline(aGraphic)(points, Color'rgb(1.0,0,0), LineStyle)
  call end_drawing(aGraphic)

  let location_x(aGraphic) = 350   ''shift position of this shape

  ''3) draw a polymark.  Use the same points as above
  create aGraphic
  file this aGraphic in graphic_set(theView)
  let name(aGraphic) = "green polymark"
  call begin_drawing(aGraphic)
  let type(MarkStyle) = MarkStyle'_diamond
  let size(MarkStyle) = 600
  call draw_polymark(aGraphic)(points, Color'rgb(0,1,0), MarkStyle)
  call end_drawing(aGraphic)

  let location_x(aGraphic) = 700   ''shift position of this shape

  ''4) draw an arc.
  create aGraphic
  file this aGraphic in graphic_set(theView)
  let name(aGraphic) = "cyan arc"
  call begin_drawing(aGraphic)
  let pattern(LineStyle) = LineStyle'_solid
  call draw_arc(aGraphic)(150, 150, 50, pi.c / 4, 3 * pi.c / 4,
2.7 Icons

The Icon object is derived from the Graphic object and is used to graphically represent any moving or static object inside a window. Like the Graphic objects, the Icon is composed of a group of shapes such as lines, polygons, text. In addition, an Icon object has an appearance property that can be loaded in from SimStudio. The SimStudio Icon Editor’s JPEG image import feature allows an Icon object to be shown by a raster image. This allows photographs and images created by other programs to be shown in a SIMSCRIPT III application. Another important feature of the Icon object is its ability to have motion over simulation time. A velocity can be defined for the icon, which will cause its position to be updated automatically as simulation time is advanced.

2.7.1 Creating and loading an Icon

An Icon object will appear in the canvas of a Window object. It must be filed into the graphic_set owned by a View object that is in turn filed into the view_set owned by a Window object. An appearance for the Icon object can be constructed in the SimStudio Icon Editor. In this case a template whose name matches the name saved under the Icon Editor can be assigned to the appearance property of the Icon object.
2.7.2 Background Icons

At this point, a distinction should be made between a background icon and a dynamic icon. A background icon is static and not specifically intended to be animated. Usually the goal is for the background to appear with the same size and position in the program as it does in the SimStudio Icon editor. For background icons the following steps should be performed in the Icon editor:

1. From the “Icon properties” dialog (Edit/Icon properties… menu), make sure that the “Xlow, Xhi, Ylow, Yhigh” values match the coordinate system of the view that the Icon object is to be attached to.

2. Under “Center Point” the “Automatic recenter” check box should be cleared and both X and Y values should be set to “0”. When using this method it is usually not necessary to assign the location_x and location_y properties of the Icon object at runtime.

For example, suppose that an icon saved under the name “big shape” should appear within the canvas of the object MyWindow. Also, assume that the shape is to be used as a static background icon under the coordinate system Xlow=100, Xhigh=200, Ylow=100, Yhigh=200.
''Example7.sim
''Displaying a background icon in a window

preamble including the gui.m subsystem
end

main

define MyWindow as a Window reference variable
define MyView as a View reference variable
define MyIcon as an Icon reference variable

create MyWindow, MyView, MyIcon

file MyView in view_set(MyWindow)    ''attach view to window
file MyIcon in graphic_set(MyView)   ''attach icon to view

     call set_world(MyView)(100, 200, 100, 200)

     ''Displaying an icon in a window
     let appearance(MyIcon) = Templates'find("big shape")

     call display(MyWindow)  ''display everything

     while visible(MyWindow) <> 0 call handle.events.r(1)  ''wait
end
2.7.3 Dynamic Icons

Dynamic icons are designed to represent either a moving object, or an object whose location is not known until runtime. Every Icon object inherits the location_x and location_y attributes from Graphic. These attributes should be assigned at runtime after the appearance attribute. The range of values for location_x and location_y depend on the coordinate system of the View object instance containing the Icon object. The display_at method will set both location_x and location_y properties before making the icon visible.

When constructing this variety of icon in SimStudio the following should be done:

1. From the “Icon properties” dialog (Edit/Icon properties… menu), make sure that the “Xlow, Xhi, Ylow, Yhigh” values match the coordinate system of the view that the icon object is to be attached to.

2. Under “Center Point”, usually the “automatic recenter” box should be checked. In the application, this will force the location_x and location_y attributes to refer to the geographic center point of the icon. If “Automatic recenter” is not checked, then the correct center point (sometimes called the hotspot) should be entered before the icon is saved.

In this example, an icon is constructed in SimStudio and saved under the name “little shape”. The rules listed above are followed when constructing this icon. Thousands of instances of this icon are created and filed into the view_set of a view with coordinates Xlow=0, Xhigh=100, Ylow=0, Yhigh=100.
''Example8.sim
''Displaying many dynamic icons in a window

preamble including the gui.m subsystem
end

main
    define MyWindow as a Window reference variable
    define MyView as a View reference variable
    define MyIcon as an Icon reference variable

    create MyWindow, MyView

    file MyView in view_set(MyWindow)
    call set_world(MyView)(0, 100, 0, 100)  ''must match "Icon Properties"
    call display(MyWindow)   ''draw the window

    ''scatter thousands of Icon objects at random locations
    for i = 1 to 10000
        do
            create MyIcon
            file MyIcon in graphic_set(MyView)  ''attach icon to view
            let appearance(MyIcon) = Templates'find("little shape")
            call display_at(MyIcon)(uniform.f(0, 100, 1), uniform.f(0, 100, 1))
            call handle.events.r(0)
        loop
        while visible(MyWindow) <> 0
            call handle.events.r(1)  ''wait for window to be dismissed
        end
    end
2.7.4 Animating an Icon object in a Simulation

The Icon object defines several attributes allowing it to be animated by SIMSCRIPT. The most common way to add animation is by assigning a velocity to the Icon. As simulation time progresses, the location is automatically updated as determined by the velocity, causing the entity to be redrawn. The \texttt{set\_speed\_course(speed, direction)} method can be called to specify both the \texttt{speed} with which the icon should travel and its direction. The speed is in View Coordinate Units per Simulated Time Unit. The direction is specified in radians and is measured counter-clockwise from the positive X axis.

For example, suppose you want to move the Icon reference variable called “MyIcon” in a north-east direction at the speed of 100 coordinate units per time unit:

\[ \text{call set\_speed\_course(100, pi.c/4 "'45 degrees (north-east')")} \]

Another way to change the velocity of an icon object is to assign the components of the velocity vector directly. The \texttt{velocity\_x} and \texttt{velocity\_y} attributes can be assigned to the speed at which the icon is moving “left/right” and the speed it is moving “up/down” respectfully.

In order for the icon to actually “move” simulation time must be advanced through the use of a “wait” or “work” statement. As simulation time is advanced, the positions of all Icon objects that have a non-zero velocity component are updated automatically.

For example, to move an icon strait up for 2 units of time:

\[ \text{Let velocity\_x(MyIcon) = 0} \]
\[ \text{Let velocity\_y(MyIcon) = 100} \]
\[ \text{Wait 2 units "'move 200 up"} \]

It is important to set the initial position of the icon before setting its velocity. The \texttt{location\_x} and \texttt{location\_y} attributes (inherited from Graphic) should be assigned before the first wait statement is encountered.
2.7.5 Simulation Time and Real Time

In a SIMSCRIPT III program, simulation time is normally advanced to the time at which the next event, process or process method is due to execute. However, when graphics are added to a simulation, large jumps ahead in simulation time are normally not desirable. This would cause the Icon objects to “jump around” the window thus giving a false impression to the user of the true nature of the motion. When GUI.M is used, SIMSCRIPT III will automatically perform “time scaling”. With time scaling, SIMSCRIPT III will increment the value of TIME.V is slowly. The position of every moving icon in the simulation is updated each whenever TIME.V is changes, thus smoothing out the motion of the Icons.

Time scaling is controlled by the global variable called TIMESCALE.V. The value of TIMESCALE.V establishes a scaling between real-time and simulation time. TIMESCALE.V refers to the number of 1/100 second intervals of real time per unit of simulation time. Setting TIMESCALE.V = 100 establishes a one-to-one mapping of simulation time units and real elapsed seconds. Therefore, decreasing the value of TIMESCALE.V has the effect of making the simulation run faster, provided there is enough computer power to do both the computational simulation and the animated graphics. There is no guarantee that this ratio of real time to simulation time will be maintained as the simulation runs. When there is not enough processing speed, additional elapsed real time will be taken, but simulation time is not affected. I.e. the results of the simulation will not be altered.

In this example, an icon will move from the lower left corner (0,0) to the upper right corner of the view with a speed of 4000 coordinates per time unit.
''Example9.sim
''shows an Icon object moving with a velocity during a simulation

preamble including the gui.m subsystem
  begin class MyIcon
    every MyIcon is an Icon and has a
    move_about process method
  end
end

process method MyIcon'move_about
  let location_x = 0
  let location_y = 0
  call set_speed_course(4000.0, PI.C / 4.0) ''move north east
  work 10 units
  call set_speed_course(0.0, 0.0) ''stop the shape
  work 5 units
end

main
  create a Window, View, MyIcon
  file View in view_set(window)
  file MyIcon in graphic_set(view)
  let appearance(MyIcon) = Templates'find("shape")
  let timescale.v = 100   ''1 second per time unit
  activate a move_about(MyIcon) now
  start Simulation
end
2.7.6 Custom animation

There may be some cases where modeling an object with a simple linear velocity is not adequate. Obtaining non-linear motion with automatic update of Icon positions is still possible to achieve. The Icon object defines methods that allow an application defined motion. The three methods typically used in this case are motion, start_moving, and stop_moving.

The motion method is called automatically whenever simulation time is advanced. If TIMESCALE.V is non zero, this means that the motion method will be called repeatedly during a WORK or WAIT statement to facilitate smooth movement of icons. To implement customized motion over time, the Icon class should be subclassed and the motion method overridden. The amount of time elapsed since the last call to motion is passed as a parameter. This can be used by the implementation of motion to compute the next position of the icon.

Calling the start_moving method will add the Icon object to a private set of moving objects. The motion method will be called periodically after the call to start_moving. Call the stop_moving method to halt the calls to motion and remove the Icon object from the set of moving objects. Note that when a non-zero velocity is assigned to the Icon object, start_moving is called implicitly. The motion method will be called periodically after assigning a velocity or calling the set_speed_course method.

In the following example program, the Icon object is subclassed and the motion method is overridden. The implementation of this override moves the Icon in an elliptical orbit.
''Example10.sim
''Using customized motion while animating Icon objects

preamble including the gui.m subsystem

begin class MyIcon
  every MyIcon is an Icon and has
  an about_x, an about_y, a theta,
  a move_in_circles process method, and
  overrides the motion
  define about_x, about_y, theta as double variables
  define move_in_circles as a process method given
  2 double arguments
end

process method MyIcon'move_in_circles(x,y)
  let about_x = x
  let about_y = y
  call start_moving
  work 25 units      '''motion'' method called during work
  call stop_moving
  wait 3 units       '''pause to see the icon stopped
end

method MyIcon'motion(dt)
  add dt to theta
  call display_at(about_x + 2000.0 * cos.f(theta),
                  about_y + 6000.0 * sin.f(theta))
end

main
  define icon1, icon2 as MyIcon reference variables
  create a Window, View, icon1, icon2
  file View in view_set(window)   '''set up the view
  call display(Window)            '''show the window
  file this icon1 in graphic_set(view)   '''set up the icons
  file this icon2 in graphic_set(view)
  let appearance(icon1) = Templates'find("shape")   '''use the "shape" icon
  let appearance(icon2) = Templates'find("shape")   '''for both
  let timescale.v = 50          '''0.5 second per time unit
  activate a move_in_circles(icon1)(8000, 16000) now
  activate a move_in_circles(icon2)(22000, 16000) in pi.c units
  start Simulation
end
3. Graphs

This chapter describes features of the SIMSCRIPT III language which support both the display of numerical information in a variety of static and dynamic chart formats, and the representation of changing values using a variety of graphs. Several classes are provided in GUI.M that can be created by the application and used to display a specially defined variable or attribute. (These classes will be referred to as “graph” objects since they are derived from a common object named Graph.) Graph objects are constructed in SimStudio, and loaded into the program at runtime.

All Graph objects contain methods that can be called to update the value shown in the graph. However, the display variables and dynamic histograms described in SIMSCRIPT II.5 are supported in SIMSCRIPT III. SIMSCRIPT III also allows for object attributes and class attributes to be declared as a DISPLAY variable or as a DYNAMIC HISTOGRAM. This means that as the attribute or histogram can be “hooked up” to a Graph object such that the magnitude shown in the graph automatically reflects the current value of the attribute. This lifts the necessity for the programmer to locate every assignment of the attribute (both direct and indirect) then make a method call to update the value shown in graph.

3.1 Graph objects

The Graph objects found in GUI.M support all graphs constructed in SimStudio. The following table summarizes these objects.

<table>
<thead>
<tr>
<th>Graph object</th>
<th>SimStudio Appearance</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>Analog clock</td>
<td>A clock showing the current time. Can be adjusted in SimStudio for 12 or 24 hour mode.</td>
</tr>
<tr>
<td></td>
<td>Digital clock</td>
<td></td>
</tr>
<tr>
<td>Meter</td>
<td>Dial</td>
<td>Show the current value of a single variable.</td>
</tr>
<tr>
<td></td>
<td>Level meter</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Digital display</td>
<td></td>
</tr>
<tr>
<td>Plot</td>
<td>2D chart</td>
<td>2-dimensional chart and can either show a histogram or a time trace plot (where TIME.V is plotted on x axis).</td>
</tr>
<tr>
<td>Piechart</td>
<td>Pie chart</td>
<td>Each slice represents a data value. It is sized based on its percentage of the sum of all slices.</td>
</tr>
</tbody>
</table>

After creating an instance of a Clock, Meter, Plot or Piechart object, the program should assign the appearance property to a Template object obtained from the Templates class. The assignment will link this instance to the graph constructed by SimStudio and saved in the graphics.sg2 file. The instance should then be filed into the graphic_set owned by a View object. Call the display method to make the graph visible, or to update it after a
change has been made. For example, suppose a “Level meter” is constructed in SimStudio and saved under the name “value 1”. The code to show the meter in a program might look something like this:

```plaintext
define myMeter as a Meter reference variable
...
create myMeter
let appearance(myMeter) = Templates'find("value 1")
file myMeter in graphic_set(myView)
call display(myMeter)
```
3.2 Meter Object: Graph a Single Variable

Setting the data value or values shown in a graph varies depending on which graph object is being used. The value shown in a Graph object can be updated directly by calling Meter\texttt{.set_value}, Plot\texttt{.plot_data}, Piechart\texttt{.set_slice}, or Clock\texttt{.set_time}. Graphs that have been properly connected to a “display variable” or dynamic histogram will be updated automatically when the variable changes.

Figure 3: Meters

The Meter object is used to show the value of a single variable, object attribute or class attribute. This variable or attribute quantity must be defined as a “display variable”. Display variables must be defined in the PREAMBLE using the following syntax:

\begin{verbatim}
DISPLAY VARIABLES INCLUDE variable1, variable2,...
\end{verbatim}

This declaration is made in addition to normal variable declarations. In other words the variables used in this statement must be known to SIMSCRIPT at the time the declaration is made. If the variables referenced above are object or class attributes, the statement must appear within the same BEGIN CLASS..END block that the attribute is declared in. Implicitly defined variables and attributes can be graphed. For example, the number of entities in a set call “queue” could be graphed even though “N.QUEUE” is not explicitly declared in the program.

The connection between the attribute and the Meter object is bridged using the SIMSCRIPT “SHOW WITH” statement. Instead of providing a text string after “WITH” as is done in SIMSCRIPT II.5, the Meter object reference can be specified. This is the generalized mechanism by which Graph objects are hooked up to display variables and dynamic histograms, i.e.

\begin{verbatim}
SHOW variable WITH graph_object_reference
\end{verbatim}
In the following example suppose a “Level meter” is constructed in SimStudio and saved under the name “queue length”. We want the level meter to show the number of items in the set named “queue” owned by the class Owner. The bar in the level meter should rise as objects are filed into the set.

''Example11.sim
''Uses a level meter to show the number of objects in a set

preamble including the gui.m subsystem
begin class Member
   every Member may belong to a queue
end
begin class Owner
   the class owns a Member's queue
   display variables include n.queue
end
end

main
   define myMeter as a Meter reference variable
   define myWindow as a Window reference variable
   define myView as a View reference variable
   define i as an integer variable

   create myWindow, myView, myMeter
   file this myView in view_set(myWindow) ''set up graphics hierarchy
   file myMeter in graphic_set(myView)
   let appearance(myMeter) = Templates'find("queue length")
   show Owner'n.queue with myMeter ''connect n.queue to myMeter
   call display(myWindow)  ''draw everything
   for I = 1 to 10000
      do
         create a Member
         file this Member in Owner'queue  ''this will update myMeter's bar
         call handle.events.r(0)
      loop
   end
3.3 Clock object: Show the Time

GUI.M provides a Clock object that can be used to show simulation time. Both analog (circular with hands) and digital clocks. The clocks are dynamic in nature and update automatically whenever the display variable (containing time) changes in value.

![Image of Clocks](image)

**Simulation Time**

Figure 4: Clocks

A *Clock* object is initialized in the same manner as any other Graph (such as the *Meter* explained previously). The *set_time* method may be called to update the time value given the hours, minutes and seconds. A *Clock* object can also be used like a *Meter*, where a single double or real variable is declared as a DISPLAY VARIABLE and graphed. This value indicates the number of “days”, and is converted automatically to the hours, minutes and seconds shown on the clock. For example:

```plaintext
define clocktime as a double variable
display variables include clocktime
```

To have *clocktime* updated as the simulation time is advanced, a time synchronization routine can be written and assigned to the *TIMESYNC.V* system variable. This routine will then be called whenever SIMSCRIPT is about to update the value of *time.v* and will allow you to update the clock by assigning your display variable to the given TIME. The clock will be updating regardless of the number of concurrent processes. The following example shows how an automatic clock can be used in the simulation.
'Example12.sim
'Shows a clock that is updated as the simulation goes forth

preamble including the gui.m subsystem
  define clocktime as a double variable
  display variables include clocktime
  processes include dummy
end
routine clock_update given time yielding newtime
  let newtime = time
  let clocktime = time  ''update the time shown by the Clock object
end
process dummy
for i = 1 to 4000
  wait 1.25 / (24. * 60.) units  ''wait 1.25 simulated minute
end
main
  create a Window, View, Clock
  file this View in view_set(Window)
  file this Clock in graphic_set(View)
  let appearance(Clock) = Templates'find("analog clock")
  show clocktime with Clock
  call display(Window)
  let timescale.v = 100 * 24 * 60  ''1 real sec. per sim. minute
  let timesync.v = 'clock_update'
  activate a dummy now
  start simulation
end
3.4 Plot object: Showing Histograms and Trace Plots

The Plot object can be used for Histograms (dynamic and static), time trace plots, and X-Y plots. A Plot object can contain one or more datasets, each representing some statistic compiled on a global variable, object attribute or class attribute. Each dataset can contain a fixed number of "cells" (bar-graph, histogram, surface chart representation in SimStudio) or can collect a new data point each time the monitored value changes (continuous representation).

A Plot object is created at runtime, but must be loaded from the graphics.sg2 file created by SimStudio. Within SimStudio, a “2D-chart” must be constructed and saved to enable a Plot object to be used.

3.4.1 Histograms

The purpose of a histogram is to show the user how often a variable or quantity takes on a particular value, or range of values. For example, a bar located at X=15 with a height of Y=30 would mean that the graphed quantity Q is equal to 15 on 30 separate occasions. (If the histogram for Q is defined in an “accumulate” statement instead of a “tally”, then “Y=30” would instead refer to the number of units of simulation time when “Q=15”.) Histogram names should not be included in a DISPLAY VARIABLES INCLUDE… statement, but are instead declared through a TALLY or ACCUMULATE statement.

```
TALLY hist_name (low TO high BY interval) AS THE DYNAMIC
      HISTOGRAM OF var_name
```

or

```
ACCUMULATE hist_name (low TO high BY interval) AS THE DYNAMIC
      HISTOGRAM OF var_name
```

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

In Figure 5 a histogram (constructed as a "2-D Chart" in SimStudio) is shown which was obtained from the example called "bank" which simulates bank customers waiting in line for a fixed number of "teller" resources. Each bar in the histogram shows the number of "customers" that waited between (n) and (n+1) minutes for a teller, where the number of minutes (n) is shown on the x-axis.
The "SHOW HISTOGRAM" statement is used to link the histogram defined in an ACCUMULATE or TALLY statement with the Plot object. If the Plot object was saved by SimStudio while containing more than one dataset, the multiple histograms can be shown in the same Plot by including their names in the same "SHOW HISTOGRAM" statement. A reference to the Plot object should be supplied after the word "WITH".

\[
\text{SHOW HISTOGRAM } \text{hist.name1, hist.name2, ... WITH plot.object.reference}
\]

The above statements must be invoked before any values are assigned to the tallied variable (and before any other reference is made to the histogram name). Assignments to the tallied variable will automatically update the bars in the Plot object.

The boundaries (low and high) specified in the "TALLY HISTOGRAM" statement should match the Minimum and Maximum values defined for the X axis in SimStudio. The "cell width" for each dataset added to the Plot in SimStudio should match the interval specified in the TALLY statement. If variable names are used for the histogram limits (low to high by interval) these will be automatically initialized from the X-axis graduations specified on the chart. If the value being tallied is an object attribute, these low, high and interval variables must be declared as class attributes of the same class. Should the displayed bounds on the Y-axis be exceeded during the simulation, the histogram will rescale automatically. The X-axis will not be rescaled. Data points whose X value is greater than the maximum of the X-axis are plotted to the last (rightmost) cell. When the X value is less than the minimum of the X-axis are plotted to the first (leftmost) cell.

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

\[
\text{ERASE HISTOGRAM name1, ...}
\]
SIMSCRIPT will allow you to show more than one histogram in the same *Plot* object. Assuming you have added multiple datasets to your plot in the 2-d Chart editor, each of these data sets is connected to one of the histogram names listed in the SHOW statement.

The following program will show two histograms in the same chart. The first histogram will show a uniform random distribution while the second shows a normal distribution. A Plot object is employed to show the histograms. A Plot is shown at the top of the window containing a single histogram, while a second plot contains 2 histograms. The Tallied data for variable “randvar1” is linked to the first Plot, while data for both “randvar2” and “randvar3” are shown in dataset #1 and #2 (respectfully) in the second Plot. Each time either of these variables is assigned, the corresponding plot is updated automatically. Notice that each SHOW statement precedes the assignments to randvar1, randvar2 and randvar3. Histogram limits for both “histo2” and “histo3” are declared in terms of variables. These variables obtain their values from either the data set cell width, or (if that value is 0) the X-axis tic interval (major) of the 2-d chart created in SimStudio.
Preamble including the gui.m subsystem
  define randvar1, randvar2, randvar3, lo, hi, delta as double variables
  tally histo1(0 to 10 by 1) as the dynamic histogram of randvar1
  tally histo2(lo to hi by delta) as the dynamic histogram of randvar2
  tally histo3(lo to hi by delta) as the dynamic histogram of randvar3
end

main
  define count as integer variables
  define plot1, plot2 as Plot reference variables
  create Window, View, plot1, plot2
  file this View in view_set(Window)
  file this plot1 in graphic_set(View)
  file this plot2 in graphic_set(View)
  let appearance(plot1) = Templates'find("histogram1")
  let appearance(plot2) = Templates'find("histogram2")
  show histo1 with plot1 ""link HISTO1 and the Plot1 object
  show histo2,histo3 with plot2 ""link histo2 and 3 with Plot2 object
  call display(Window) ""show everything
  for count = 1 to 50
    let randvar1 = exponential.f(5.0,1)
  for count = 1 to 500
    do
      let randvar2 = uniform.f(lo, hi, 1)
      let randvar3 = normal.f((hi + lo) / 2, (hi - lo) / 10, 1)
    loop
    while visible(Window) <> 0
      call handle.events.r(1)
  end

Accumulated statistics are weighted by the duration of simulated time for which the value remains unchanged. The following example uses a process method to generate the sample data, waiting for simulation time to elapse between each sample. In this example, the ACCUMULATE statement is used in the declaration instead of the TALLY statement.
"example14.sim
'Show how to ACCUMULATE a dynamic
preamble including the gui.m subsystem
begin class Holder
  every Holder has
    a randvar,
    a sample process method
  the class has
    a lo, a hi, and a delta
  accumulate hist(lo to hi by delta) as the dynamic histogram of randvar
define randvar, lo, hi, delta as double variables
end
end

process method Holder'sample
  until time.v gt 500
    do
      wait exponential.f(5.0, 1) units
      let randvar = uniform.f(lo, hi, 2)
      loop
    end
end

main
  create a Window, View, Plot, Holder
  file View in view_set(Windows)
  file Plot in graphic_set(View)
  let appearance(Plot) = Templates'find("histogram3")
  show histogram hist(Holder) with Plot
  call display(window)
  activate a sample(Holder) now
  let timescale.v = 10
  start simulation
end
3.4.2 Time Trace Plots

SimStudio will allow you to create a Plot object that shows the value of a single variable plotted on the Y-axis while simulation time is plotted on the X-axis. In this case, instead of using a histogram a display variable is used in conjunction with the Plot object. Essentially the time trace plot allows the user to see how a single variable has changed over the duration of the simulation.

Include a trace plot by using SimStudio to create a “2-D Chart”. The following changes should be made in the editor: From the "Chart Properties" dialog box, check the "trace plot" check box. Also, ensure that every dataset in the graph has the "Continuous Surface" representation.

There are two options for when simulation time becomes greater than the maximum value on the chart. The first option is to "scroll" previous data to the left thus making room for more data. When TIME.V exceeds the X-axis maximum, a constant is added automatically to both the X-axis minimum and maximum. Therefore, time value data at the beginning (right side) of the plot is discarded. If this "scrolling window" is not appropriate, check the "Compress Data" box in the "X-Axis Properties" dialog box in SimStudio. Under this option, SIMSCRIPT will not increase the X-axis minimum when TIME.V becomes too large, thereby keeping all previous data points.

In the following example, an object attribute (defined as a display variable) is linked to a Plot object and incremented after waiting a random amount of time. The attributes history can be seen in the plot.

![Figure 6: Time trace plot](image-url)
''Example15.sim
''Using a Plot object for a time trace of a single variable

preamble including the gui.m subsystem

begin class Holder
every Holder has
  a var,
  a sample process method
display variables include var
define var as a double variable
end
end

process method Holder'sample
until time.v gt 300
do
  wait exponential.f(5.0, 1) units
  add 1 to var   ''this will automatically update the Plot
loop
end

main
  create a Window, View, Plot, Holder
  let color(Window) = Color'_white
  file View in view_set(Window)
  file Plot in graphic_set(View)
  let appearance(Plot) = Templates'find("trace plot")
call display(window)
  show var(Holder) with Plot
  activate a sample(Holder) now
  let timescale.v = 10 '' 0.1 sec per unit

  start simulation
end
### 3.4.3 X-Y Plots

The Plot object can be used to generate a simple line, surface, or bar graph without using display or histogram variables. The `Plot.plot_data` method can be called explicitly to either add a point to a “continuous surface” graph or change an existing bar in a bar, discrete surface or histogram type graph. (See the “Data set representation” section on the “Data Set Properties” dialog in the 2-d Chart editor). The `Plot.plot_data` method takes as parameters the data set number and the X and Y coordinates of the data point. (Data sets are numbered starting from one.) The `Plot.clear_data` method can be called to eliminate all data in the Plot.

In this example the function $y = \frac{100}{x+1}$ is plotted. There are no display variable or histogram declared, and the SHOW statement is not needed.

```
''example16.sim
''Plotting data without a "display" or "histogram" variable
    Preamble including the gui.m subsystem
end

main
    create Plot, View, Window
    file View in view_set(Window)
    file Plot in graphic_set(View)
    let appearance(Plot) = Templates'find("x-y plot")
        for x = 1.0 to 100.0
            call plot_data(Plot)(1, x, 100.0 / x)   ''add data to the plot
            call handle.events.r(1)
        end
    call clear_data(Plot)
end
```

Bar charts and discrete interval surface charts can also be used to show the elements of an array declared as a DISPLAY VARIABLE. The value of each individual element of the array is plotted to the Y value of a bar in the chart. The index of an element is mapped not to the X axis, but to the cell number. Cells in this case are numbered from the left starting at “1”. The data set representation should be either “Bar graph”, “Discrete surface”, or “Histogram”. This example shows the elements of an object array attribute being graphed in real time using a bar graph.
begin class Engine
  every Engine has a piston_y,
  and a piston process method
  define piston as a process method given 1 integer argument
  define piston_y as a 1-dim double variable
  display variables include piston_y
end
define rpm=20 as a constant  "rotations per minute"
end
process method Engine\'s piston\(number\)
  define s as a double variable
  let s = time.v
  while time.v < 100  "run for 100 minutes"
    do
      let piston_y\(number\) = 2. + 2. * cos.f(_rpm * (time.v-s) * 2. * pi.c)
      work 0.001 units
    loop
end
main
create Plot, View, Window, Engine
reserve piston_y(Engine) as 8
file View in view_set(Window)
file Plot in graphic_set(View)
let appearance(Plot) = Templates\'find("engine")
show piston_y(Engine) with Plot
end
3.4.4 Setting up the X and Y axes

Sometimes the range of values to be plotted is not known at compile time. For this reason, the Plot object provides methods that allow the X and Y axis numbering, tic mark intervals and boundaries to be set at runtime. The following table shows Plot object properties relating to the axes that can be used on the left side of an assignment. All methods are of mode “double”.

<table>
<thead>
<tr>
<th>X Axis property</th>
<th>Y Axis property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>min_x</td>
<td>min_y</td>
<td>Minimum value shown on the axis</td>
</tr>
<tr>
<td>max_x</td>
<td>max_y</td>
<td>Maximum value shown on the axis</td>
</tr>
<tr>
<td>interval_x</td>
<td>interval_y</td>
<td>Major tic mark interval. For data sets with a zero cell width, interval_x is used as the width.</td>
</tr>
<tr>
<td>num_interval_x</td>
<td>num_interval_y</td>
<td>Numbering interval measured in the coordinates defined by the axis.</td>
</tr>
<tr>
<td>grid_interval_x</td>
<td>grid_interval_y</td>
<td>Grid line interval</td>
</tr>
<tr>
<td>minor_interval_x</td>
<td>minor_interval_y</td>
<td>Smaller tic mark intervals.</td>
</tr>
<tr>
<td>intercept_x</td>
<td>intercept_y</td>
<td>Intercept_x is the point along the X axis where the Y axis crosses. Similarly for intercept_y</td>
</tr>
</tbody>
</table>
4. Forms

Many times an application will require a robust set of tools for user interaction. Users may expect to control the whole application through a menu-bar displayed at the top of the window, or to use a smaller context menu to dynamically make changes to an individual item. Dialog boxes are used as a convenient way to communicate data quantities to a program. GUI.M provides Form objects that include menu bars, dialog boxes, tool bars (palette), and popup message boxes. SIMSCRIPT supports these objects using JAVA which allowing programs to be recompiled and executed on different operating systems without recoding or retooling the graphical user interface. All objects derived from the Form object are portable from platform to platform.

4.1 Using Form objects

The DialogBox, MenuBar and Palette objects are derived from Form. Dialog boxes are windows containing a variety of input controls including buttons, text labels, tabular controls, single and multi-line text, combo, value, list, radio, and check boxes. Menu bars can appear at the top of the canvas of a Window object. A Palette is attached to the top, left, right or bottom of a Window object and contains rows of buttons that can be dragged onto the Window’s canvas.

All objects derived from Form (with the exception of MessageBox) must be constructed using the SimStudio Dialog box, Menu bar or Palette editors. As with Graph and Icon objects, the Form object defines an appearance property that can be assigned a Template object. The Templates’ find method returns a Template given the name with which it was saved in the SimStudio editor. Every Form object instance must be attached to a Window object. This is accomplished by filing the Form instance into the form_set owned by Window.
4.2 DialogBox object

The DialogBox object is derived from the Form object and represents both modal and modeless dialogs. Dialog boxes provide an interactive way for the user to enter input data. A dialog box is a window containing a variety of input controls including buttons, text labels, tabular controls, single and multi-line text, combo, value, list, radio, and check boxes. Tabbed dialogs can also be created in SIMSCRIPT. A dialog must be constructed in the SimStudio dialog box editor and loaded into the application as described above. The DialogBox object supports both modal and modeless interaction.

In a modeless dialog execution does not stop when the object is displayed. A modeless dialog box should be constructed in the SimStudio dialog box editor by un-checking the “modal interaction” box in the “Dialog box Properties” dialog. If a modeless interaction is needed, the display method should be used to make the dialog visible. Modal dialogs are used when the application needs to wait for the user to make a selection from the dialog before continuing. The DialogBox class defines an accept_input method that can be called to display the dialog and wait for user interaction.

Modal interaction is halted when the user presses a “terminating” button, at which accept_input returns. A button in the dialog can be marked as terminating through the “Button properties” dialog. (Usually, the “Cancel” and “OK” buttons are terminating.) Another check box in the “Button properties” dialog called “Verifying” can be useful for dialog boxes that contain numerical fields (i.e. value boxes). Pressing a “Verifying” button will range check values in all value boxes in the dialog. If a value is out of range, characters “<<<” will be shown after the value the dialog will “beep”. Usually “OK” buttons are marked as verifying.

In the following example a dialog box is constructed in SimStudio and saved under the name “simple dialog”. The dialog contains a single “Verifying” button that can be pressed to dismiss it.

```
''Example18.sim
''Displaying a simple dialog box
Preamble including the gui.m subsystem
End
main
create DialogBox, Window
call display(Window)

'assume a dialog named "simple dialog" has been created in SimStudio
let appearance(DialogBox) = Templates'find("simple dialog")

'attach the dialog box to this window
file this DialogBox in form_set(window)

'show the dialog. This will not return until user clicks on button
call accept_input(DialogBox)
end
```
4.3 Using Field objects for data transfer

The *Field* object provides an interface for passing data back and forth between the executing program and an item (check box, text box, etc) in a form. *Field* objects must be filed into the *field_set* owned by Form. In most cases, the *Field* object is created and filed into this set automatically at the time the *appearance* attribute is assigned. A corresponding field is created for each item in a dialog box, each menu item in a menu bar, and each button in a palette.

The *name* attribute of the field is initialized to the “Field name” given to the corresponding item in the SimStudio editor. This allows the program to obtain a reference to a particular field given its *name*. The *find* method defined by the *Form* class does just that. Given the text “name” of a field, *Form.find* will return the *Field* reference value for the item.

The *Field* object defines several properties that can be used on the left or right side of an assignment. When the property is assigned, the text, value or selection state shown by the corresponding item in the dialog box will change to show the new value. When the property is “read”, the current text or value typed in by the user will be returned by the method. The table below describes many different data types and dialog box items that can be.
<table>
<thead>
<tr>
<th>Method</th>
<th>Data type</th>
<th>Dialog Items</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
<td>text</td>
<td>tree item</td>
<td>Sets the name of the icon shown with the item. (Without the “.jpg” extension).</td>
</tr>
<tr>
<td>selected</td>
<td>integer (0 or 1)</td>
<td>button, check box, tree item, radio button</td>
<td>Gets/sets the current selection state of an item that can be toggled. '1' means selected, while '0' means unselected.</td>
</tr>
<tr>
<td>selected_at</td>
<td>integer (0 or 1)</td>
<td>list box, radio box, table</td>
<td>Gets/sets the current selection state of an item in the dialog composed of a list of selectable cells or buttons. The first argument refers to the row number of the item or button starting from “1”. For tables, row '0' refers to the column headers, and the second argument refers to a column number. A value of '1' assigned to this field means to select the item, while a value of '0' means to de-select it.</td>
</tr>
<tr>
<td>string</td>
<td>text</td>
<td>combo box, menu item, text box, tree item, label</td>
<td>Gets/sets text data in the field.</td>
</tr>
<tr>
<td>string_at</td>
<td>text</td>
<td>list box, table</td>
<td>Gets/sets text item in a field given the row and column of the item.</td>
</tr>
<tr>
<td>strings</td>
<td>1-dim array of text</td>
<td>combo box, list box, multi-line box, table</td>
<td>Gets/sets an array of text data in the field. If used on the left, the items in the list shown in the dialog will be replaced by the text in the array.</td>
</tr>
<tr>
<td>value</td>
<td>double</td>
<td>label, progress bar, value box</td>
<td>Gets/sets the numerical value shown in an item in a dialog box. For labels, the “Formatted Real” button in the “Label properties” dialog should be checked.</td>
</tr>
</tbody>
</table>

The `accept_input` method of the `DialogBox` object will return with the `Field` reference pointer for the terminating button that was selected to dismiss the dialog. The `name` attribute of this Field can be inspected to determine which button was clicked on. Note that if the user dismisses the dialog box by clicking on the “X” in the window frame, `accept_input` will return with “0”.

In this example a dialog box is shown that contains several different fields. Assigning the `Field` object properties before the dialog is displayed initializes dialog box items. The `Form/Find` method is used here to obtain the `Field` object reference pointer.
''Example19.sim
''Displaying a dialog box containing many different fields

Preamble including the gui.m subsystem
End

main

  define items as a 1-dim text array
  define db as a DialogBox reference variable

  ''create window, dialog box and load it from a template
  create db, Window
  call display(Window)
  let appearance(db) = Templates'find("field dialog")
  file this db in form_set(window)

  ''initialize the fields in the dialogs before showing it
  ''set the check box to "checked" state
  let selected(find(db)("check_box")) = 1

  ''set text and value boxes
  let string(find(db)("text_box")) = "Initial text"
  let value(find(db)("value_box")) = 60.25

  ''set the items in the multi line edit box
  reserve items(*) as 3
  let items(1) = "The quick brown fox"
  let items(2) = "jumped over the"
  let items(3) = "lazy dog's back."
  let strings(find(db)("edit_box")) = items(*)
  release items(*)

  ''set cell row 2, column 1 in the table
  let string_at(find(db)("table")(2,1)) = "Cell text"

  ''select the second radio button
  let selected_at(find(db)("radio_box")(2,0)) = 1

  ''show the dialog.  Wait for a terminating button to be pressed
  let field = accept_input(db)

  ''now read the fields and print them out
  if field <> 0 and name(field) = "ok"
    write selected(find(db)("check_box")) as "Check box: ", I 2, /
    write selected_at(find(db)("radio_box")(2,0)) as "2nd Radio button: ", I 2, /
    write string(find(db)("text_box")) as "Text box: ", T *, /
    write value(find(db)("value_box")) as "Value box: ", D(8,2), /
    let items(*) = strings(find(db)("edit_box"))
    for i = 1 to dim.f(items(*))
      write i, items(i) as "Edit box line ", I 3, ": ", T *, /
    always
  end
4.4 Event Notification

In some cases, you will want to provide code to immediately handle Form object input events generated by the user (such as button clicks, text box modification, menu selection etc) while the simulation is running. Whenever a user clicks on a button or makes a change to a dialog box item, the `Form:action` method is called automatically. An application requiring this immediate notification can subclass the `Form` object and override its `action` method. This is a useful way to perform immediate validation or cross-checking of fields. The action method is defined as follows:

```plaintext
define action as an integer method given
  1 FormEvent reference argument
```

The single argument to the action method is an instance of the `FormEvent` object. The “field” attribute of this object contains a reference pointer to the `Field` object that was clicked on or changed. The `name` attribute of this `Field` object can be compared against the field names of known items. The “id” field of the `FormEvent` object contains a constant indicating one of several varieties of event. The following table lists the possible values for “id”, the meaning of each value, and which of the other attributes of the `FormEvent` object are used.

<table>
<thead>
<tr>
<th>Id</th>
<th>FormEvent Attributes</th>
<th>Dialog Items</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_button_dropped</td>
<td>drop_x, drop_y, drop_view, field</td>
<td>palette button</td>
<td>Indicated that a palette button marked in the SimStudio “Palette Button Properties” dialog as “Draggable” was dragged from the palette and dropped. drop_x, drop_y indicate the Canvas coordinates of the drop location. drop_view is the View object containing the drop point.</td>
</tr>
<tr>
<td>_button_pushed</td>
<td>field</td>
<td>button menu item, table cell, tree, item</td>
<td>One of the items was clicked on by the user.</td>
</tr>
<tr>
<td>_close</td>
<td>-----</td>
<td>-----</td>
<td>Indicated that the user has clicked on the “X” in the corner of a dialog box’s window.</td>
</tr>
<tr>
<td>_data_changed</td>
<td>field</td>
<td>check box, combo box, radio button, text box, value box</td>
<td>This event is sent when data being shown in a dialog item has been changed by the user. The return key must be pressed after entering data into a text/value box. The “Selectable using return” check box must be marked in the “Value box properties” or “Text box properties” dialog in the SimStudio dialog box editor.</td>
</tr>
</tbody>
</table>
On of the following constants defined in the Form class should be returned from the 
action method overridden by the subclass.

_continue - Accept the input and continue.
_terminate - Terminate the interaction and return from the accept_input method.

In this example, a dialog box containing several different items is displayed. The
DialogBox object is subclassed and the action method overridden. Code in the action
method displays information about the item that was clicked on or changed by the user.

''Example20.sim
''Showing how the "action" method can be overridden

Preamble including the gui.m subsystem
begin class TestDialogBox
  every TestDialogBox is a DialogBox and
  overrides the action ''called when user acts on an item
end

method TestDialogBox'action(event)
  if field(event) <> 0
    let string(find("sel_name")) = name(field(event))
    always
      select case id(event)  ''determine which event has happened
        case FormEvent'_button_pushed
          case FormEvent'_data_changed
            ''display the data contained by the event’s field
            let selected(find("sel_state")) = selected(find(name(field(event))))
            let string(find("sel_text")) = string(find(name(field(event))))
            let value(find("sel_value")) = value(find(name(field(event))))
          endselect
        endcase
        case FormEvent'_close
          write as "User has closed the dialog!", /
          return with _terminate  ''return from accept_input()
        endcase
        return with _continue  ''keep going
      endselect
  end
end

main
  ''create window, dialog box and load it from a template
  create TestDialogBox, Window
  call display(Window)
  let appearance(TestDialogBox) = Templates'find("action dialog")
  file this TestDialogBox in form_set(window)

  ''show the modal dialog
  call accept_input(TestDialogBox)
end
4.5 Enable and Disable fields

In many cases, you may want to activate and deactivate items in your dialog box in response to the user changing one of the fields. When the activated property of the Field object is assigned a value of ‘0’, the corresponding item in the dialog box or menu bar is deactivated and can no longer be selected or edited by the user. The item will not disappear but instead appear greyed out. For example, to deactivate the field named “my check box field” in the dialog referenced by “my_dialog”:

Define check_field as a Field reference variable
Let check_field = find(my_dialog)("my check box field")
let activated(check_field) = 0

It is also possible to hide and show fields using the SIMSCRIPT "display" and "erase" methods inherited from the GuiItem object. i.e.

call erase(field) ’’make the field disappear
call display(field) ’’make field visible

This example shows how typically the activated property is used. While the user is making changes to items in the dialog, the correct activation state of related items can be maintained by overriding the action method.
''Example21.sim
''Using the "activate" property
''Clicking on radio buttons will change which control is enabled

Preamble including the gui.m subsystem
begin class TestDialogBox
   every TestDialogBox is a DialogBox and
   overrides the action ''called when user acts on an item
end
end

''override the action method so that items can be deactivated while the
''program is waiting in accept_input

method TestDialogBox''action(event)
   if field(event) <> 0
      select case name(field(event))
         case "use_text"
            let activated(find("text_box")) = 1
         case "use_value"
            let activated(find("text_box")) = 0
            let activated(find("value_box")) = 1
         default
            let activated(find("value_box")) = 0
      endselect
      always
      return with _continue   ''keep going
   end
end

main
   ''create window, dialog box and load it from a template
   create TestDialogBox, Window
   call display(Window)
   let appearance(TestDialogBox) = Templates'find("activate dialog")
   file this TestDialogBox in form_set(window)

   ''initially, deactivate the value box
   let activated(find(TestDialogBox)("value_box")) = 0

   ''show the modal dialog
   call accept_input(TestDialogBox)
end
4.6 Trees

One of the items that can be added to a dialog in the SimStudio editor is the tree. A tree contains a list of items that can be viewed hierarchically, with items containing other items. Each item in the tree consists of a label and an optional jpeg image, and each item can have list of sub items associated with it. By clicking an item at runtime, the user can expand and collapse the associated list of sub items.

Using SimStudio you can specify the initial set of items and sub items in the tree. However, most applications will need to set up the tree at runtime. The hierarchy shown in the tree can be constructed at runtime by creating instances of Field objects and filing them appropriately into the field_set owned by the tree field. Filing a Field into a field_set is the equivalent of adding a new item to the tree. The string attribute of the Field is the label of the tree item. For example, to add the items labelled “One” and “Two” to a tree field named “little tree”:

```plaintext
define field1, field2 as Field reference variables
create field1, field2
let string(field1) = "One"
let string(field2) = "Two"
file this field1 in field_set(find("little tree"))
file this field2 in field_set(find("little tree"))
```

If we wanted the item labeled “Two” to contain items labeled “A” and “B” the additional code could be added:

```plaintext
define field3, field4 as Field reference variables
create field3, field4
let string(field3) = "A"
let string(field4) = "B"
file this field3 in field_set(field2)
file this field4 in field_set(field2)
```

An item in a tree can be shown with a small image next to it. The image property of the field object can be assigned the name of a file containing a bitmapped image. This image should be small enough to fit into the list of items and in be JPEG format. (A typical size is usually 16x16 or 24x24 pixels). The name should be specified without the “.jpg” extension. The following example shows how an application can populate a tree with a hierarchy of items. JPEG images are shown with the items, and the application assumes that the files “I.jpg”, “II.jpg” and “III.jpg” exist.
'Example22.sim
'Using a tree in a dialog box
Preamble including the gui.m subsystem
end

main
define tree, field1, field2, field1_1, field1_2, field1_2_1
    as Field reference variables

    'create window, dialog box and load it from a template
    create DialogBox, Window
call display(Window)
    let appearance(DialogBox) = Templates'find("tree dialog")
    file this DialogBox in form_set(window)

    let tree = find(DialogBox)("tree")
create field1, field2, field1_1, field1_2, field1_2_1

    'set the label and image file name for each item in the tree
    let string(field1) = "Level 1 First Item"
    let string(field2) = "Level 1 Second Item"
    let string(field1_1) = "Level 2 First Item"
    let string(field1_2) = "Level 2 Second Item"
    let string(field1_2_1) = "Level 3 First Item"
    let image(field1) = "I"    'assume "I.jpg" exists
    let image(field2) = "I"
    let image(field1_1) = "II"    'assume "II.jpg" exists
    let image(field1_2) = "II"
    let image(field1_2_1) = "III"    'assume "III.jpg" exists

    'build the hierarchy
    file this field1 in field_set(tree)
    file this field2 in field_set(tree)
    file this field1_1 in field_set(field1)    'field1 will contain 2 items
    file this field1_2 in field_set(field1)
    file this field1_2_1 in field_set(field1_2)    'field1_2 will contain 1

    'initially select the item at the lowest level
    let selected(field1_2_1) = 1

    'show the modal dialog
call accept_input(DialogBox)
end
4.7 Tables

A table is an item in a dialog composed of a two dimensional arrangement of selectable text fields or "cells". The table can be scrolled both horizontally and vertically. All cells in the same column have the same width, but you can define the width of this column. A table can have both column and row headers. The headers are fixed and will remain in view when the table is scrolled.

The end user can navigate through a table using the left-, right-, up- and down-arrow keys. The action method is invoked whenever a cell is clicked on or an arrow key is used to move to focus on a different cell. The table can be set up to automatically add a new row of cells at the bottom when the user attempts to move below the last row. Use SimStudio to add a table to a dialog box.

As with other items, a field object will be created automatically for each table in a dialog. The field's string_at method can be used on the left to specify the text of a particular cell. The arguments to this method are the row and column of the cell in the table. If the table is constructed in SimStudio to use column headers (on the top) the text of a header is assigned by specifying "0" as the row number argument to the string_at method. Similarly, "0" can be specified as the column number to assign a text value to a row header.

Another way to specify the cell text values is to put all the text into a 1-dim array and assign the array to the strings method. The array must be reserved big enough to hold all the cells. Text values are laid out in row major order. For example, assume the table has '.NUM_COLUMNS' columns (including row headers) and '.NUM_ROWS' rows. The index into the array for cell (COLUMN,ROW) is computed as follows:

```
let table_values((ROW-1) * .NUM_COLUMNS + COLUMN) = "hello"
```

The action method of the DialogBox object containing the table can be overridden to handle events generated by the table. The selected_at method can be called to get or set the currently selected cell in the table.

In the following example, a table constructed in SimStudio is populated with data at runtime. In this example the column headers are set in the SimStudio DialogBox editor, while the row headers are set in the program.
'Example23.sim

'Using a table

Preamble including the gui.m subsystem
define _header=0, _name_col, _company_col, _occupation_col as constants
end

main
define table_field as a Field reference variable

'create window, dialog box and load it from a template
create DialogBox, Window
call display(Window)
let appearance(DialogBox) = Templates'find("table dialog")
file this DialogBox in form_set(window)

let table_field = find(DialogBox)("table")
let string_at(table_field)(1, _header) = "1"
let string_at(table_field)(1, _name_col) = "Zapp Brannigan"
let string_at(table_field)(1, _company_col) = "DOOP"
let string_at(table_field)(1, _occupation_col) = "Starship Captain"
let string_at(table_field)(2, _header) = "2"
let string_at(table_field)(2, _name_col) = "Mom"
let string_at(table_field)(2, _company_col) = "Mom's Old Fashion Robot Oil"
let string_at(table_field)(2, _occupation_col) = "President/CEO"
let string_at(table_field)(3, _header) = "3"
let string_at(table_field)(3, _name_col) = "Morbo"
let string_at(table_field)(3, _company_col) = "Channel Radical 2 News"
let string_at(table_field)(3, _occupation_col) = "News Anchor Monster"
let string_at(table_field)(4, _header) = "4"
let string_at(table_field)(4, _name_col) = "Richard M. Nixon"
let string_at(table_field)(4, _company_col) = "Government"
let string_at(table_field)(4, _occupation_col) = "President of Earth"
let string_at(table_field)(5, _header) = "5"
let string_at(table_field)(5, _name_col) = "Dr. Zoidberg"
let string_at(table_field)(5, _company_col) = "Planet Express"
let string_at(table_field)(5, _occupation_col) = "Medical Doctor"

' show the modal dialog
call accept_input(DialogBox)
end
4.8 Menu bars

In many typical applications, the user interacts with a menu bar attached to the top of the window. In many cases the entire range of functionality is accessible through the menu bar. Given its widespread use, most users expect to be able to control the program through the menu bar.

A menu bar is composed of several menus arranged in a row on a bar across the top of a window frame. Clicking on one causes its menu-pane to be displayed. Clicking on an item inside a menu causes it to be selected. Cascading menus are supported, meaning that menus can contain other menus and so on.

The menu bar is implemented by creating a menubar object. Like other object derived from Form, a menu bar is constructed in SimStudio and loaded into the program by assigning the appearance attribute. At this time a field object is created for each menu in the menu bar. A field object is filed in to the field_set owned by the menu or menu item that contains it. A menu bar is always modeless and is shown by calling its display method. The application must subclass the menubar object and override the action method to receive notification of menu item selection while the simulation is running.

A program can dynamically set and clear check marks next to any menu item. To display the check mark, set the selected attribute of the menu item field to "1". Clear the mark by setting the attribute to "0". Before the check mark is actually drawn or erased from the menu, the corresponding Field object must be re-displayed by calling the display method.

In the following example, a menubar is created in SimStudio consisting of two menus. One of the menus contains a submenu that allows the user to change the background color of the window.
begin class MyMenuBar
every MyMenuBar is a MenuBar, and
has a current_color, and
overrides the action
define current_color as a text variable
end
end

method MyMenuBar'action(event)
if current_color <> ""
    let selected(find(current_color)) = 0 ''erase menu item check
always
let current_color = name(field(event))
let selected(find(current_color)) = 1   ''show menu item check
''see which menu item was selected by comparing the field name
select case name(field(event))
    case "black"   let color(Window) = Color'_black
    case "white"   let color(Window) = Color'_white
    case "red"     let color(Window) = Color'_red
    case "green"   let color(Window) = Color'_green
    case "blue"    let color(Window) = Color'_blue
    case "cyan"    let color(Window) = Color'_cyan
    case "magenta" let color(Window) = Color'_majenta
    case "yellow"  let color(Window) = Color'_yellow
    case "exit"    return with _terminate
endselect
call display(Window)
return with _continue
end

main
''create window, dialog box and load it from a template
create Window, MyMenuBar
create Window
let appearance(MyMenuBar) = Templates'find("menu bar")
file this MyMenuBar in form_set(Window)
call display(MyMenuBar)    ''display the menu bar
''wait til user closes window or uses the exit menu
while visible(MyMenuBar) <> 0 and visible(Window) <> 0
call handle.events.r(1)
end
4.9 Palettes

Toolbars and palettes are also popular components in a user interface. (For the sake of SimStudio, we will refer to either component as a *palette.*) A palette is basically a horizontal or vertical bar containing a row (or column) of buttons, with each button showing a little picture image. An application may need a palette at the top of the window to provide quick access to commonly used items. Many applications also allow users to “drag” the image shown on a palette button onto the canvas of the window. A palette can be attached to any edge of the window. The buttons contained on a palette can be typical push buttons, or can toggle (stay down when pressed.)

Palettes are implemented with the *Palette* object which is derived from the *Form* object. Palettes must therefore be constructed in SimStudio and loaded into the program by assigning a Template to the appearance attribute of the Palette object instance.

Buttons marked in SimStudio as “draggable” (using the “Palette Button Properties” dialog) enable drag and drop. In this case your program is notified of the action allowing you to create a display entity representing the object that was dragged. To get the notification of a drag event, the Palette object must be subclassed and the action method overridden. The *id* attribute of the *FormEvent* argument will be _button_dropped_. The *drop_x*, *drop_y* and *drop_view* attributes can be used to determine the drop location (in canvas coordinates) and the *view* containing the point at which the button was dropped.

In the following example, a Palette object is constructed in the SimStudio Palette Editor and displayed in a window. When the user drags buttons from the palette to the canvas an *icon* object is created and placed in the drop location.
begin class TestPalette  
  every TestPalette is a Palette,  
  overrides the action  
end  
end

'override the action method to get notification of palette button  
'clicks and drops
method TestPalette'action(event)  
  define my_icon as an Icon reference variable
  select case id(event)  
    case FormEvent'_button_pushed  
      write name(field(event)) as T *, ", was pressed", /  
    case FormEvent'_button_dropped  
      if drop_view(event) <> 0  
        'put a new icon in the drop location  
        create an my_icon  
        let appearance(my_icon) = Templates'find(name(field(event)) +  
          ".icn")  
        file this my_icon in graphic_set(drop_view(event))  
        call display_at(my_icon)(drop_x(event), drop_y(event))  
      always  
      default  
    endselect  
  return with _continue
end
main
  create Window, TestPalette, View
  file this TestPalette in form_set(Window)
  file this View in view_set(Window)
  let appearance(TestPalette) = Templates'find("palette")
  call display(Window)
  while visible(Window) <> 0  
    call handle.events.r(1)
end
4.10 PopupMenu Objects

Popup menus allow an end user to right click in the canvas of a window to display a small menu showing a list of choices. They are sometimes referred to as *context menus* because a different menu can easily be displayed depending on both where and when the user right-clicks in the window.

A popup menu is implemented through GUI.M by creating a `PopupMenu` object. To construct a popup menu in SimStudio, use the Menu bar editor to create a menu bar. The first (leftmost) menu in this menubar will be used as the popup while all other menus are ignored. The name given to the menubar will be passed in the application to the `Templates\'find` method. The corresponding Template is then assigned to `appearance` attribute of the `PopupMenu` object. For example, assume the menu bar is saved under the name “shape popup”:

```plaintext
create a PopupMenu
file this PopupMenu in form_set(Window)
let appearance(PopupMenu) = Templates\'find("shape popup")
```

If the items to be contained in the menu are not known until runtime, the popup menu can be constructed by program code. This can be done as follows:

1. create a `field` object for each item to be placed in the menu.
2. Assign the name used to identify the item to the `name` attribute of the field.
3. Assign the label shown on the item to the \textit{string} attribute of the field.
4. File each field in the \textit{field\_set} owned by the \textit{popupmenu} object.

For example, to create a popup menu with two items, one labeled “Start” and the other labeled “Stop”:

\begin{verbatim}
define field1, field2 as Field reference variables
create a PopupMenu
create a field1, field2
let string(field1) = "Start"
let string(field2) = "Stop"
let name(field1) = "start_picked"
let name(field2) = "stop_picked"
file this field1 in field_set(PopupMenu)
file this field2 in field_set(PopupMenu)
file this PopupMenu in form_set(Window)
\end{verbatim}

The \textit{accept\_input} method is used to show a popup menu. The method will block execution until the user selects an item. A reference to the selected \textit{Field} object is returned from \textit{accept\_input}. The application can then inspect the \textit{name} attribute of this field to determine the proper action. If the user does not make a selection (i.e. the \texttt{<escape>} key is pressed before an item is picked) the zero is returned by \textit{accept\_input}.

In the following example program two objects called \textit{StyleGraphic} and \textit{ColorGraphic} subclass the \textit{Graphic} object. Each object overrides the \textit{action} method to display a different popup menu when an instance of the graphic is clicked on with the right mouse button. The popup menu for \textit{StyleGraphic} allows the user to change the fill style while the menu for \textit{ColorGraphic} allows the color to be changed. The popup menu for \textit{StyleGraphic} is constructed in SimStudio and the menu for \textit{ColorGraphic} is constructed in the program.
Using a popup (context) menu

begin class StyleGraphic
  every StyleGraphic is a Graphic,
  overrides the action
end

begin class ColorGraphic
  every ColorGraphic is a Graphic,
  overrides the action
end

```
'' the method is called whenever the ColorGraphic is clicked on
method ColorGraphic' action(event)
  define color as an integer variable
  define field1, field2, field3 as Field reference variables

  if button_number(event) > 1  ''right mouse button pressed
    create a PopupMenu
    create a field1, field2, field3
    let string(field1) = "Red"
    let string(field2) = "Green"
    let string(field3) = "Blue"
    let name(field1) = "red"
    let name(field2) = "green"
    let name(field3) = "blue"
    file this field1 in field_set(PopupMenu)
    file this field2 in field_set(PopupMenu)
    file this field3 in field_set(PopupMenu)
    file this PopupMenu in form_set(Window)

    let Field = accept_input(PopupMenu)
    if Field <> 0
      '' see which field in the popup menu was selected
      select case name(field)
        case "red"    let color = Color'_red
        case "green"  let color = Color'_green
        case "blue"   let color = Color'_blue
        default
      endselect

      '' draw the solid square using the new color
      let pattern(FillStyle) = FillStyle'_solid
      call begin_drawing(ColorGraphic)
      call draw_rectangle(ColorGraphic)
        (17000, 10000, 10000, 10000, color, FillStyle)
      call end_drawing(ColorGraphic)
      call display(ColorGraphic)
    always
    always
    return with Graphic' action(event)
end
```
'the method is called whenever the StyleGraphic is clicked on
method StyleGraphic'event)
  if button_number(event) > 1 "'right mouse button pressed
  create a PopupMenu
  file this PopupMenu in form_set(Window)

  "'load a menu bar created in SimStudio. Its first menu will be
  'used as the popup menu
  let appearance(PopupMenu) = Templates'find("style popup")

  "'show the popup menu then wait for user to click on an item.
  let Field = accept_input(PopupMenu)
  if Field <> 0
    "'see which field in the popup menu was selected
    select case name(field)
      case "hollow"
        let pattern(FillStyle) = FillStyle'_hollow
      case "solid"
        let pattern(FillStyle) = FillStyle'_solid
      case "narrow_diagonal"
        let pattern(FillStyle) = FillStyle'_narrow_diagonal
      case "medium_diagonal"
        let pattern(FillStyle) = FillStyle'_medium_diagonal
      case "wide_diagonal"
        let pattern(FillStyle) = FillStyle'_wide_diagonal
      case "narrow_crosshatch"
        let pattern(FillStyle) = FillStyle'_narrow_crosshatch
      case "medium_crosshatch"
        let pattern(FillStyle) = FillStyle'_medium_crosshatch
      case "wide_crosshatch"
        let pattern(FillStyle) = FillStyle'_wide_crosshatch
      default
        endselect

    "'draw the red square using the new fill style
    call begin_drawing(StyleGraphic)
    call draw_rectangle(StyleGraphic)
      (5000, 10000, 10000, 10000, Color'_red, FillStyle)
    call end_drawing(StyleGraphic)
    call display(StyleGraphic)
    always
    always

    return with Graphic'event)
  end
main
create Window
let title(Window) = "Popup Menu Test"
call display(Window)

create View
file this View in view_set(Window)

'create object derived from Graphic, draw a red square.
create StyleGraphic
file this StyleGraphic in graphic_set(View)
call begin_drawing(StyleGraphic)
call draw_rectangle(StyleGraphic)(5000, 10000, 10000, 10000, Color'_red, 0)
call end_drawing(StyleGraphic)
call display(StyleGraphic)

'create object derived from Graphic, draw a green square.
create ColorGraphic
file this ColorGraphic in graphic_set(View)
call begin_drawing(ColorGraphic)
call draw_rectangle(ColorGraphic)
   (17000, 10000, 10000, 10000, Color'_green, 0)
call end_drawing(ColorGraphic)
call display(ColorGraphic)

'wait for the window to be dismissed
while visible(Window) <> 0
   call handle.events.r(1)
end
4.11 MessageBox Objects

In some cases you will want only to display a simple message that allows the user to answer "yes", "no", or "cancel". Toolkits provide built in dialog boxes just for that purpose. The MessageBox object allows you to add these built in message boxes to your program to make use of these dialogs.

![MessageBox Object](image)

Figure 8: MessageBox object.

A MessageBox object can be constructed in SimStudio using the “message box editor”. It is loaded in from the “.sg2” file in the same way as the DialogBox object- by assigning a Template to the appearance attribute. A MessageBox differs from a DialogBox object in that it does not contain data “fields”. A message boxes is more simplified than a dialog box and does not allow the user to enter information. Instead, it will contain one of the following sets of response buttons:

- a) OK button only
- b) OK and Cancel buttons
- c) Yes and No buttons
- d) Yes, No and Cancel buttons
- e) Retry and Cancel buttons
- f) Abort, Retry and Ignore buttons

Any one of these buttons can be designated in SimStudio as the default button. This button is activated when the user presses the <return> key. Message boxes come in different styles, with each style showing a different icon in it the dialog. The following five styles are available:

- a) Plain
- b) Stop Sign
- c) Question
- d) Exclamation
- e) Information

All MessageBox objects are modal and the accept_input method should be called to display the dialog and wait for user input. The interaction is ended and the dialog disappears when the user clicks on any button. The accept_input method will return with
an integer indicating which button was pressed. The constants \_ok\_button, \\_cancel\_button, \_yes\_button, \_no\_button, \_abort\_button, \_retry\_button, \_ignore\_button are possible return values.

Normally, the text of the message can be entered in SimStudio when constructing the dialog. If this text is not known until runtime, the MessageBox object provides two attributes that can be assigned to set the text to be displayed. A single line message can be assigned to the message\_line attribute. Longer messages should be placed into a 1-dim text array and assigned to the message\_lines attribute. (each element of the array will show as a line of text in the dialog).

A messagebox object can be constructed at runtime by assigning the style, responses, and default\_button properties. The default\_button property is assigned to one of the "button" constants listed above. The style property basically controls which icon is displayed in the dialog box. Options are \_plain\_message, \_stop\_message (stop sign), \_question\_message (question mark), \_alert\_message (exclamation), and \_information\_message (Circle-I).

For responses the constants available are: \_ok\_response, \_ok\_cancel\_response, \_yes\_no\_cancel\_response, \_yes\_no\_response, \_retry\_cancel\_response, \_abort\_retry\_ignore\_response.

The following program displays two MessageBox objects. The first is constructed entirely within the SimStudio message box editor. The multiple lines of text shown in the second box are assigned at runtime.
4.12 FileBox object

A FileBox object allows the user to browse the hard drive(s) for a file with a particular extension. It works like the messagebox described above – An instance of a filebox object is instanciated, initialized, filed into the form_set owned by a parent window, and the accept_input function method is called to show the browser. Assign the filter attribute to specify the extension of files to be shown using "*" as a wildcard. For example, a filter value of "*.dat" would have the dialog shown only files that end in " .dat". The path attribute can be assigned to the initial directory.

Accept_input returns ‘1’ if the user has successfully selected a file, and ‘0’ if the “cancel” button was pressed. At this point the file attribute will have the name of the file selected by the user.
main
    define browser as FileBox reference variable
    create window
    call display(window)

    create browser
    let filter(browser) = "*.jpg"  ''only show jpeg files
    file this browser in form_set(window)
    ''show the browser dialog. Wait for user to press "select"
    if accept_input(browser) <> 0
        write file(browser) as "The user selected the file ", T *, /
    else
        write as "No file was selected", /
    always
end
5. 3d Graphics

The Simscript III 3d graphics package is an object-oriented wrapper around the OpenGL toolkit. The public preamble 3d.m.sim contains the basic class descriptions that allow the programmer to create windows containing 3d graphical scenes. “Camera” and “light” objects can be created to implement realistic scenes in an object-oriented fashion.

Hierarchical scene-graphs allow the programmer to specify graphical objects (called “nodes”) as containing other objects (positioning and orientation of a “container” node will automatically affect the position and orientation of the sub-nodes it contains. Each node in the scene graph represents a visible graphic in the window. The node is implemented with the 3dnode class.

In order to see the nodes in the scene-graph a “Camera” is necessary. Each 3d program must create at least one camera and orient it properly in order to view the scene-graph. Multiple cameras are allowed, and there viewports can be overlapped or tiled on the canvas of a 3d window. A camera can also be attached as part of a scene-graph allowing it to move or rotate with its parent node. A 3dcamera class implements the camera.

Multiple light sources are allowed in a Simscript 3d graphics program. A “Light” can be created a positioned and oriented with respect to the scene. Light objects are also part of the scene-graph and are implemented using the 3dlight class. A “Model” represents a specific shape that can be displayed multiple times in the same window at different locations and orientations. Methods of this class allow geometry and properties for various surfaces and lines to be specified. “Graphic” items are attached to the scene-graph that reference “model” objects. Models are implemented with the 3dmodel class.

A “Material” provides a covering of 3d surfaces. The material can have a distinct color and shininess. Texture mapping is supported via the material object. The texture is stored in a 2d raster image file such as a Windows Bitmap or (“.BMP”) file or a Targa Graphics file (“.TGA”). 2d Coordinates specified with the 3d vertices identify the position in the raster image file that is to map to the surface. Surface materials are implemented with the 3dmaterial class.

A “Graphic” is a node in the scene-graph that provides the base class for all visible 3d objects whose geometry is defined at runtime. Sub-classes of the graphic object are provided that support various types of lines and surfaces. The graphic object can also be sub-classed and its “draw” method overridden. This allows customized surface types to be created and rendered by the application. The graphic is implemented with the 3dgraphic class.

The “World” is used as the root of the scene-graph. Several worlds can appear in the same window allowing such things as control panels, graphs, heads up displays, or any other such visual aids that are separate from the 3d scene, but appear in the same window.
Graphics in multiple worlds will not interleave but overlap regardless of the distance from viewer. The \texttt{3dworld} class is used to implement a world.

Nodes contained in the scene can have animated motion that is linked to elapsing simulation time.

\section*{5.1.1 A Typical 3d Graphics Program}

A typical 3d graphical simulation will display many graphical objects, which may or may not be moving with respect to simulation time. Light sources and cameras must be created. To create such a program, the following steps should be performed:

1) Write customized drawing code (if necessary).
   a) Define sub-classes of \texttt{3dgraphic} in the preamble and override the “draw” method.
   b) In the implementation of “draw” make calls to \texttt{begin\_drawing} and \texttt{end\_drawing}.
   c) Between \texttt{begin\_drawing} and \texttt{end\_drawing}, make calls to \texttt{3dgraphic} class methods to define the vertices, normal vectors, texture coordinates and materials of the shape.

2) Write event handling code (if necessary).
   a) If keyboard input, mouse input, or window resize handling is needed, define in the preamble a subclass of \texttt{3dwindow} and override its “action” method.
   b) Implement the “action” method by writing code to inspect the attributes of the “\texttt{3devent}” instance passed as its argument and perform the necessary processing.

3) Create windows and worlds.
   a) For each window that is needed, create an instance of a \texttt{3dwindow} object. Call methods to set size, position and title.
   b) Create instances of the \texttt{3dworld} object. Usually only one world is needed, unless you are implementing a control panel, heads up display, or the application requires multi-layered graphics.
   c) File \texttt{3dworld} instances into the “world\_set” owned by the \texttt{3dwindow} object.
   d) If a hierarchy (or groupings of objects) is needed, create instances of the \texttt{3dnode} object. File them into the node\_set owned by the \texttt{3dworld} in which they are to be used. They can otherwise be filed into the node\_set owned by another node.

4) Create camera(s)
   a) For each view of the scene, create an instance of a \texttt{3dcamera} object.
b) Set the location and orientation of the camera so that it is some distance from the scene, but is pointing at the scene. If necessary, call other methods to customize the view.

c) File instances into the “camera_set” owned by the 3dworld showing the view.

d) A 3dcamera can also be attached to a 3dnode allowing its position and orientation to be determined by the parent node. If necessary, the 3dcamera object can ALSO be filed into a “node_set”.

5) Create light(s) (if necessary).
   a) For each light illuminating the scene, create an instance of a 3dlight object.
   b) Set the location of the light appropriately. For spot lights it may be necessary to set the orientation also.
   c) File instances into the “light_set” owned by the 3dworld showing the view.
   d) A 3dlight can also be attached to a 3dnode allowing its position and orientation to be determined by the parent node. If necessary file the light into a “node_set”.

6) Create models (if necessary).
   a) For each unique model that is used in the 3dworld, create a instance of a 3dmodel object.
   b) If the model geometry is stored in a 3dStudio (.3ds) file, a AutoCAD (.dxf) file or a SIMGRAPHICS II (.sg2) file, call the “read” method to read the file.
   c) File each 3dmodel instance in the “model_set” owned by the 3dworld.

7) Create materials.
   a) Create an instance of a 3dmaterial object for each unique color or texture used in the program. (Materials defined offline (in a .3ds file) are created automatically when 3dmodel’read is called.)
   b) Set the color, shininess, and texture_name attributes.
   c) File each 3dmaterial instance into the “material_set” owned by 3dworld.

8) Create other graphical objects for the simulation.
   a) Create instances of sub-classes of 3dgraphic such as 3dfaces, 3dlines, 3dpoints, to represent objects in the simulation. Create instances of the subclasses of 3dgraphic described in step 1.
   b) Create instances of 3dnode to represent a model on-screen. Assign the “model” attribute so that the 3dnode will draw this model at its location.
   c) Some objects may have a “material” attribute that needs to be assigned.
   d) File these instances into the “node_set” owned by the 3dworld, or by another 3dnode.

9) Call the display method of the 3dwindow. This will show the window.
10) Before starting the simulation, activate the “3dwindow’animate” method. This method will keep refreshing the window as often as possible as the simulation runs. Don’t forget to set “timescale.v”.

5.1.2 Example code for a typical 3d graphics program

''Example29: Example of a initializing a 3d graphics application
''Requires ford.3ds model file
preamble including the 3d.m subsystems
begin class my_window
every my_window is a 3dwindow and
overrides the action
end

define the_car as a 3dnode reference variable
end

''2) Write event handling code overriding the 3dwindow'action method
method my_window'action(event)
if id(event) = 3devent'_close
stop
always
if id(event) = 3devent'_key_down
select case key_code(event)
  case 3devent'_right_key
   call rotate_y(the_car)(2)
  case 3devent'_left_key
   call rotate_y(the_car)(-2)
  case 3devent'_up_key
   call rotate_x(the_car)(2)
  case 3devent'_down_key
   call rotate_x(the_car)(-2)
  case 3devent'_home_key
   call move(the_car)(0.0, 0.0, 5.0)
  case 3devent'_end_key
   call move(the_car)(0.0, 0.0, -5.0)
  default
endselect
always
return with 0
end

main
define window as a my_window reference variable
define world as a 3dworld reference variable
define camera as a 3dcamera reference variable
define light as a 3dlight reference variable
define model as a 3dmodel reference variable

''3) Create windows, worlds, and groups
create window
let title(window) =
  "Use the arrow, home, and end keys to move the model"
create world
file this world in world_set(window)

''4) create camera(s)
create camera
call set_orientation(camera)(0.0, 0.0, -1.0, 0.0, 1.0, 0.0)
call set_location(camera)(0.0, 0.0, 100.0)
file this camera in camera_set(world)

'5) create light(s)
create light
let ambient_color(light) = color'rgb(0.4, 0.4, 0.4)
let diffuse_color(light) = color'rgb(1.0, 1.0, 1.0)
call set_location(light)(0.0, 500.0, 500.0)
file this light in light_set(world)

'6) create model(s)
create model
call read(model)("ford.3ds", "")
file this model in model_set(world)

'7) create material(s)
'(materials for this example are created automatically when
' model is read)

'8) create objects used in the simulation
'' a) Create instances of 3dnode to represent the model on-screen.
'' Assign the "model" attribute to link the model to the 3dnode.
create the_car
let model(the_car) = model
file this the_car in node_set(world)

'9) Call the display method of the 3dwindow.
''This will show the window.
call display(window)

'10) Before starting the simulation, activate the 3dwindow'animate
''method. This method will keep refreshing the window as often as
''possible as the simulation runs. Don’t forget to set timescale.v.
activate a animate(window)(10000) now

let timescale.v = 100
start simulation
end
5.1.3 Class hierarchy

There are many classes that can be used to implement 3d graphics the full hierarchy for the ones found in the `3d.m` subsystem is shown in Figure 10. Figure 11 shows classes found in `3dshapes.m`. 
Figure 10: 3d.m classes

Figure 11: 3dshapes.m classes
5.2. The “scene-graph”

To write a SIMSCRIPT 3d graphics program, you must make use of “windows”, “worlds” and “nodes”. The nodes are used to compose the hierarchy of the “scene-graph”. Every 3d graphics program must construct at least one scene-graph to represent the 3d objects and background. Each “node” in this graph is represented with an instance of a 3dnode object. The 3dnode object is filed into a node_set, which is owned by another “parent” 3dnode.

5.2.1 The “3dworld”

The root of a scene-graph must always be an instance of a 3dworld object. The 3dworld class also owns a node_set for child nodes to be filed into. Every 3dworld instance is filed into a world_set owned by a 3dwindow object. The 3dwindow object represents GUI window and can be moved and resized like other windows on the computer screen. It can contain multiple 3dworlds and acts as the root of all scene-graphs to be displayed on its canvas. Basically, the rule is that every object that is to be made visible (with the exception of the 3dwindow) must be filed into an appropriate set or it will not be shown. The figure below shows a diagram of a scene-graph.

![Diagram of a scene-graph](image)

Figure 12: A typical scene graph for a tank and a battlefield

The code for creating this scene-graph would look something like this:

Define window as a 3dwindow reference variable
Define battlefield, control_panel as a 3dworld reference variable
Define tank, body, turret, housing, cannon, terrain, ground, obstacles
As 3dnode reference variables

... create window, battlefield, control_panel
create tank, body, turret, housing, cannon, terrain, ground, obstacles

file this housing in node_set(turret)
file this cannon in node_set(turret)
file this turret in node_set(tank)
file this body in node_set(tank)
file this tank in node_set(battlefield)
file this ground in node_set(terrain)
file this obstacles in node_set(terrain)
file this terrain in node_set(battlefield)
file this battlefield in node_set(terrain)
file this control_panel in world_set(window)

The 3dworld also acts as a container for other objects that are employed by nodes in the attached scene-graph. The table below lists the various sets owned by a 3dworld object:

<table>
<thead>
<tr>
<th>Set name</th>
<th>Member</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>camera_set</td>
<td>3dcamera</td>
<td>cameras used to show the scene-graph.</td>
</tr>
<tr>
<td>light_set</td>
<td>3dlight</td>
<td>light sources in the scene.</td>
</tr>
<tr>
<td>material_set</td>
<td>3dmaterial</td>
<td>materials used by 3dnodes in the scene-graph.</td>
</tr>
<tr>
<td>model_set</td>
<td>3dmodel</td>
<td>models used by nodes in the scene-graph.</td>
</tr>
<tr>
<td>node_set</td>
<td>3dnode</td>
<td>forms the scene-graph.</td>
</tr>
</tbody>
</table>

5.2.2 Setting the location of a node

Properties of the 3dnode are its location and orientation. The location_x, location_y, and location_z attributes represent the x, y, and z coordinates of the node’s position with respect to its owner node in the scene-graph. Right-handed use of these attributes is allowed, but the set_location method should be used to set these attributes. Suppose the simulation was to show a ferry shuttling passengers. Some of the code that might be used to build the scene-graph and position the objects is:

```
' put all passengers 'on the ferry' by filing
' into the node set owned by the_ferry instance
For I = 1 to 100
    File this passenger(i) in node_set(the_ferry)

' set z location of the ferry. Passengers will move with the ferry!
let location_z(the_ferry) = 1000.0

' set a passengers position with respect to ferry
call set_location(passenger(2))(2.5, 0.0, 34.9)
```

5.2.3 Setting the orientation of a node

The orientation of a node can be defined by 2 vectors, “forward” and “up” (See Figure 13). The forward vector indicated the direction of the local z-axis relative to the owner
node in the scene-graph. By default, the forward vector is (0.0, 0.0, 1.0) which would point a node in the same direction as its owner node. The “up” vector is the local y-axis relative to the node’s owner in the scene_graph. By default, this is (0.0, 1.0, 0.0). The local x-axis is computed automatically by taking the cross product of these vectors.

Figure 13: Forward and Up vectors. (In this case the positive X axis would point away from the viewer).

The attributes forward_x, forward_y, forward_z, up_x, up_y and up_z can be used on the right, but should not be assigned individually. Both forward and up vectors must always be normalized and orthogonal to each other. The set_orientation method allows both the forward and up vectors to be assigned. For example, set_orientation could be used to point a passenger in the opposite direction of the “ferry”.

```
Call set_orientation(passenger(2))(0.0, 0.0, -1.0, 0.0, 1.0, 0.0)
  ''forward_x, forward_y, forward_z, up_x, up_y, up_z
```

Another method called set_forward allows the forward direction to be specified alone, while the “up” vector is computed automatically. This vector is computed in such a way that its projection onto the positive y-axis is maximized. (for 3dcameras, this will prevent the view from tilting assuming the “floor” of the scene lies in the x-z plane)

```
define set_forward as a method given
  3 double arguments  ''forward_x, forward_y, forward_z
```

Another method that may be useful for setting the orientation of a 3dnode is the aim method. Calling the aim method will set the forward direction of the node so that it “points at” a given location. The location should be in global coordinates; non-local to the 3dnode. (The 3dworld’get_location method can be used to convert a location from local to global coordinates). When aim is used on a sub-component, the (local) orientation (forward and up vectors) of the sub-component will be modified so that the sub-component points (with its positive z-axis) at the given location. The aim method can only be called after the 3dnode has been filed into the node_set. Since the aim method uses the orientation and location of parent nodes, it should be called after all the location of all parent (grand-parent, etc.) nodes have been initialized.

```
'make a passenger turn to look at a spot on the shoreline
Call aim(passenger(2))(45000, 20.0, -20000)
```
5.2.4 Shifting the position of a node

The set_orientation and set_forward methods expect vectors that are oriented with respect to the node’s owner in the scene-graph (or the 3dworld if the 3dnode is filed in 3dworld\'node_set). Likewise, the set_location method provides coordinates with respect to the coordinate system defined by node’s owner in the scene-graph. However, in some cases it may be easier to position the node with respect to its own coordinate system. The move method will shift the position of a node by a movement right, up and forward with respect to its own axes. The node’s location will be moved along its local x-axis, y-axis, and z-axis by the three given values.

```
'\'move passenger(2) 10 units forward
call move(passenger(2))(0.0, 0.0, 10.0)
'\'dx, dy, dz
```

5.2.5 Rotating a node

It is also possible to “spin” a 3dnode on one of its three local axes. The rotate_x, rotate_y, and rotate_z methods will do just that. Each method takes an angle (in degrees) as an argument and spins the 3dnode by that amount about the local (not owner) axis. These methods are similar to the move method in that they take “delta” values instead of absolute values. For example, if an airplane is pointed forward along its positive z axis, call the rotate_x method will pitch up or down. In this case the local Y and Z axes are rotated, but the X axis will remain unchanged. Calling the rotate_y method will yaw about its y axis. Calling the rotate_z method will “roll” the airplane.

```
define rotate_x, rotate_y, rotate_z as method given
1 double argument       ''angle in degrees
```

The local axes of a node are rotated with the node itself. For example, in Figure 10, a box is first rotated about the z-axis the moved by 100.0 units in the “Y” direction (up).
5.2.4 Scaling a node

The *scale* method will modify size of the node. A scaling factor is provided for each axis and, as with *move* and *rotate* scaling is performed along the local axes. Each axis is scaled by the given scale factor (a value of “1.0” will not change the axis). Consider a process method to simulate an animated explosion. The *scale* method is called in a loop to animate the size change of the explosion. The wait statement allows a small amount of time to elapse.

```plaintext
Process method explosion'explode
    define I as a integer variable
    for i = 1 to 100
        do
            call scale(1.1, 1.1, 1.1) ''sx, sy, sz (width, height, depth)
            wait 0.1 units
        end
    loop
end
```

5.2.5 Complete Scene-graph example

In the following example, a scene-graph will be created showing some traffic cones and a tank. The tank will maneuver through the cones as the simulation runs.

```plaintext
''Example 30:
Preamble including the 3d.m, 3dshapes.m subsystem
begin class tank
    every tank is a 3dnode and has
    a speed,
    a spin_rate,
    a movement process method,
    overrides the motion
    define speed, spin_rate as double variables
end
define window as a 3dwindow reference variable
end

process method tank'movement
    define spin, forward as double variables
    define _spin_speed = 20, _forward_speed=2 as constants
```
open unit 1 for input, name is "example2.dat"
use unit 1 for input

let eof.v = 1
while eof.v = 1 and visible(window) <> 0
do
  read spin, forward
  
  ' rotate the tank
  let spin_rate = _spin_speed * sign.f(spin)
  wait abs.f(spin) / _spin_speed units
  let spin_rate = 0
  
  ' move the tank forward
  let speed = _forward_speed
  wait abs.f(forward) / speed units
  let speed = 0.0
loop

close unit 1
end

'our motion method will be called automatically as simulation time advances
method tank'motion(dt)
call 3dnode'motion(dt)
call rotate_y(dt * spin_rate)
call move(0.0, 0.0, dt * speed)
end

main
define world as a 3dworld reference variable
define camera as a 3dcamera reference variable
define light as a 3dlight reference variable
define tank_model, cone_model as a 3dmodel reference variable
define background as 3dnode reference variables
define the_tank as a tank reference variable
define cones as a 2-dim 3dnode reference variable
define _num_rows_cones=11, _num_cols_cones=11 as a constant
define i,j as an integer variable

create the window
create window
let title(window) = "Example 2: using a simple 3dnode scene-graph"

create the world
create world
file this world in world_set(window)

'point it down the negative z direction
create camera
call set_perspective(camera)(90.0, 1.0, 5.0, 115.0, 1)
call set_forward(camera)(0.0, 0.0, -1.0)
call set_location(camera)(0.0, 4.0, 60.0)
file this camera in camera_set(world)

create a light source and point it in the same direction as the camera
create light
call set_forward(light)(0.0, 0.0, -1.0)
file this light in light_set(world)
'create a background node to hold the objects
create background
file this background in node_set(world) 'add to "root" of scene-graph

'create the one model for the tank
create tank_model
call read(tank_model)("tank.3ds", "")
file tank_model in model_set(world)

'create the tank node
create the_tank
let model(the_tank) = tank_model
file the_tank in node_set(world) 'add to "root" of scene-graph
file the_tank in motion_set

'create the model for the cone
create cone_model
call read(cone_model)("cone.3ds", "")
file this cone_model in model_set(world)

'create many cones and space them out
reserve cones as _num_cols_cones by _num_rows_cones
for i = 1 to _num_rows_cones
    for j = 1 to _num_cols_cones
        do
            create cones(i,j)
            let model(cones(i,j)) = cone_model 'cones share the same model
            call set_location(cones(i,j))(i * 10 - 55.0, 0.0, j * 10 -55.0)
            file this cones(i,j) in node_set(background) 'add node to scene-graph
        loop
    loop

call display(window) 'bring up the main window and show everything
activate a movement(the_tank) now 'our process method
activate a animate(window)(10000) now 'tell window to show animation
let timescale.v = 10 '10/100 real seconds per unit of time
start simulation
end
Figure 15: Example 30 – A tank maneuvering through traffic cones.
5.3 Cameras and Lights

Every 3d graphics program must have at least one camera to view the scene-graph. A “light” is also needed to illuminate the scene-graph. Cameras are implemented with the “3dcamera” class while lights are represented by the “3dlight” class. Basically, setting up each camera involves:

a. Creating an instance of a 3dcamera object.
b. Set the location and direction of the camera so that the scene-graph will be in view by calling 3dcamera’s set_location and 3dcamera’s set_forward.
c. Set the view angle, aspect ratio, and near and far clipping planes by calling 3dcamera’s set_perspective.
d. Set the portion of the window canvas that the view from the camera should encompass by calling 3dcamera’s set_viewport.
e. File the camera in the camera_set owned by the 3dworld that the camera is to see.
f. If the camera is to be part of the scene-graph, file it into the node_set owned by a parent 3dnode.

As an example, the initialization code for the camera in the above “example #2 is as follows:

```cpp
'create camera, place it along positive z axis but
'point it down the negative z direction
create camera
call set_perspective(camera)(90.0, 1.0, 0.1, 1000.0, 1)
call set_forward(camera)(0.0, 0.0, -1.0)
call set_location(camera)(0.0, 4.0, 60.0)
file this camera in camera_set(world)
```

5.3.1 Camera location and orientation

Since the 3dcamera is derived from 3dnode, it inherits the methods that allow orientation and position to be specified. These methods are:

```cpp
set_location(x,y,z)
    'sets the location_x, location_y and location_z attributes. If
    'the camera is attached to a scene-graph, this location is with
    'respect to the parent node.
set_orientation(fx, fy, fz, ux, uy, uz)
    'set the forward vector <fx,fy,fz> and the up vector <ux,uy,uz>
set_forward(fx,fy,fz)
    'sets the forward vector <fx,fy,fz>. The camera will "point" in
    'this direction
aim(x,y,z)
    'causes the camera to point at the given target location.
    'The target location is specified in global "world"
    'coordinates
rotate(ax,ay,az,degrees)
    'rotates the camera about the given axis by the given number
    'of degrees.
```
rotate_x(degrees), rotate_y(degrees), rotate_z(degrees)
'spins the camera about its local x, y, or z axis by the
given number of degrees

As and example we will analyze how the view was set up in Example 2. In this case, the
“world” was oriented with respect to the viewer as follows:

a. Its positive X axis is pointing to the right.
b. Its positive Y axis (up vector) is pointing straight up.
c. Its and its positive Z axis is pointing at the viewer.

The default “up” vector is to point straight up, so it is not necessary to set this—in other
words calling the set_forward method is sufficient. If the positive Z axis is to point
toward the viewer, the camera should point in the opposite direction, or along the
negative Z axis.

call set_forward(my_camera)(0.0, 0.0, -1.0)

Figure 16: Camera orientation for Example 30

Now that we have defined how the coordinate axes are oriented with respect to the
viewer, the position of the camera must be specified. But there must first be some idea of
how big the scene-graph is in terms of coordinate space—so that we know how far away
from the scene to place the camera from the scene. We must also know where the scene
is in the coordinate space.

In Example 2, the scene is geographically centered at the (0,0,0) point. And since our
camera is pointing along the negative Z axis, it must be placed along the positive Z axis
with respect to the scene-graph. The scene is specified in meters. The tank moves about
an area which is about 100 by 100 meters and is located in the x-z plane. Placing the
camera about 10 meters beyond the boundaries of the lot will allow most of the scene to
be in view. Since the lot is centered about (0,0,0) we place the camera at z=+60. Also, in
order to see the traffic cones that are far away, the camera should be placed a few meters
up in the air.
5.3.2 Setting up the viewing plane

The next consideration is for setting up the camera’s projection. Two methods are available for doing this: \texttt{set\_perspective} and \texttt{set\_orthographic}. Usually, you will want to use a perspective projection. The \texttt{set\_perspective} method sets the camera’s near and far clipping planes, as well as the aspect ratio of its width to height and the angle of view.

\begin{verbatim}
define set_perspective as a method given
1 double argument, "perspective_angle > 0.0, < 180.0 degrees
1 double argument, "perspective_ratio.
2 double arguments, "perspective_near, perspective_far
1 integer argument "perspective_autosize
\end{verbatim}

The \texttt{perspective_angle} parameter is specified in degrees and represents that angle of the field of view in the \texttt{y} direction (see Figure 17). The \texttt{perspective_ratio} is the ratio of the width of the viewing plane to its height. Values below 1.0 will cause the image to appear compressed vertically. \texttt{perspective_ratio} values greater than 1 will compress the view horizontally. However, if the \texttt{perspective_autosize} parameter is “1”, the perspective ratio will be determined based on window canvas size. In this case the \texttt{perspective_ratio} argument will be ignored. Basically, turning on the \texttt{perspective_autosize} will avoid “compressing” the scene either vertically or horizontally regardless of how the user sizes the window.

For setting up the camera in Example 2, a perspective angle of 90 degrees is a good start. The camera is located at \texttt{z+60} with the nearest edge of the parking lot at \texttt{z+50} meters. A \texttt{near} clipping plane of \texttt{+5} is adequate. The farthest edge of the lot is at \texttt{z-50} meters, a distance of 110 meters. The \texttt{far} clipping plane of \texttt{+115} will allow the entire lot and tank to be seen. To avoid distorting the aspect ratio, we will pass “1” to the \texttt{perspective_autosize} argument.

\begin{verbatim}
call set_perspective(camera)(90.0, 1.0, 5.0, 115.0, 1)
\end{verbatim}
The use of the \textit{set\_orthographic} method is not common. When viewing an orthographic projection, there is no depth information provided. In other words, an object that is far away from the camera will be rendered to be the same size as when it is close to the camera. The arguments to \textit{set\_orthographic} define the clipping volume—or the coordinate boundaries of the box that encloses everything that we want to view. These units are relative to the location of the camera.

\begin{verbatim}
define set_orthographic as a method given
  2 double arguments,   ''left,   right
  2 double arguments,   ''bottom, top
  2 double arguments,   ''far,    near
  1 integer argument   ''1= adjust size automatically after window resize
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{orthographic_diagram.png}
\caption{Using the \textit{set\_orthographic} method.}
\end{figure}

It would be possible to modify Example 30 to use an orthographic transformation. The call to “set\_perspective” is replaced with a call to the “set\_orthographic” method. To simplify things we will place the camera in the center of the parking lot and allow it to “see” 50 meters to its left, its right, its front and its back.

\begin{verbatim}
create camera
call set_orthographic(camera) (-50, 50, 0, 100, -50, 50, 1)
call set_location(camera) (0.0, 0.0, 0.0)
call set_forward(camera) (0.0, 0.0, -1.0)
file this camera in camera_set(world)
\end{verbatim}

\subsection*{5.3.3 Adding a 3dcamera to the scene-graph}

Since the 3dcamera object is derived from \textit{3dnode} it can optionally be part of the scene-graph. If a camera is filed into the \textit{node\_set} owned by a parent node, it will move and rotate with the parent automatically. If for example you wanted to see the view out of the locomotive of a moving train, the 3dcamera object could be filed into the \textit{node\_set} of the locomotive’s \textit{3dnode}. The camera would then be located and oriented with respect to
the locomotive and not the ground. This would naturally show the view from the train as it moves and turns along its track.

Suppose we wanted to modify Example 2 to show the view from the “tanks” perspective instead of a static view from the parking lot. Adding the following code after the tank is created could do this:

```plaintext
file camera in node_set(the_tank)
call set_location(camera)(0.0, 4.0, 0.0) ''location relative to tank's center
call set_forward(camera)(0.0, 0.0, 1.0) ''point camera in same dir. as tank
```

(Example 3 contains the code to do this).

### 5.3.4 Viewports and multiple cameras

By default, the view seen by each camera will encompass the entire canvas of the window to which it is attached. However, the `set_viewport` method may be called to assign a rectangular box to which the view is mapped.

```plaintext
define set_viewport as a method given
  2 integer arguments,    ''x, y in pixels.
  2 integer arguments,    ''width, height pixels.
  1 integer argument      ''1=>size viewport to window canvas
```

The dimensions of the view are given in pixels with (0,0) located at the lower left corner of the window canvas (see Figure 19). Having viewports enables the use of multiple camera objects. The view from each camera can be mapped into a separate area of the canvas.

![Figure 19: Using the set_viewport method](image)

Since the viewport dimensions are specified in pixels, in order to know what the width and height parameter values to `set_viewport` should be we usually need to know the dimensions of the window canvas. This can be obtained by reading the `canvas_width`
and `canvas_height` attributes of the `3dwindow` class. Unfortunately these values are not updated until the window becomes visible. Therefore the window must be displayed before the viewports are initialized.

The last parameter to the `set_viewport` method is a flag to indicate if the viewport dimensions should be automatically adjusted as the window is resized. The viewports are modified so that they occupy the same percentage of window space after the resize operation. As such, if zero is passed for this flag and the user resizes the window to make it very small, part of the viewport may disappear.

Suppose that we want to modify Example 30 to show the view from multiple camera objects. We will create a second camera and attach it to the tank itself. The view from the tank is to appear on the left side of the canvas while the view from the parking lot appears on the right side. Basically we will be setting up each camera as in Example 30, but will be adding the code

```plaintext
call set_viewport(tank_camera)(0, 0, canvas_width(window)/2, canvas_height(window), 1)
call set_viewport(lot_camera)(canvas_width(window)/2, 0, canvas_width(window)/2, canvas_height(window), 1)
```

### 5.3.5 Tracking a moving object with the 3dcamera

Another attribute of the `3dcamera` called `tracked_node` can be assigned allowing the camera to automatically “track” or point at another node in the scene-graph. In other words, the tracked node will always appear in the center of the camera's viewport regardless of its position. As the node is tracked, the camera will align its local “up” vector with the global Y axis (in order to keep the view from “tilting”). Example 30 can easily be modified to allow the camera to track the moving tank by adding one line of code:

```plaintext
Let tracked_node(lot_camera) = the_tank
```

The fully “updated” version of Example 30 (Example 31) is below:

```plaintext
'Example 31, updated 'tank' program
Preamble including the 3d.m, 3dshapes.m subsystem
begin class tank
    every tank is a 3dnode and has
    a speed,
    a spin_rate,
    a movement process method,
    overrides the motion
    define speed, spin_rate as double variables
end
define window as a 3dwindow reference variable
end
process method tank'movement
```
define spin, forward as double variables
define _spin_speed = 20, _forward_speed=2 as constants

open unit 1 for input, name is "example2.dat"
use unit 1 for input

let eof.v = 1
while eof.v = 1 and visible(window) <> 0 do
  read spin, forward
  '' rotate the tank
  let spin_rate = _spin_speed * sign.f(spin)
  wait abs.f(spin) / _spin_speed units
  let spin_rate = 0

  ''move the tank forward
  let speed = _forward_speed
  wait abs.f(forward) / speed units
  let speed = 0.0
loop

close unit 1
end

''our motion method will be called automatically as simulation time
'advances
method tank'motion(dt)
call 3dnode'motion(dt)
call rotate_y(dt * spin_rate)
call move(0.0, 0.0, dt * speed)
end

main
define world as a 3dworld reference variable
define tank_camera, lot_camera as 3dcamera reference variables
define light as a 3dlight reference variable
define tank_model, cone_model as a 3dmodel reference variable
define background as 3dnode reference variables
define the_tank as a tank reference variable
define cones as a 2-dim 3dnode reference variable
define _num_rows_cones=11, _num_cols_cones=11 as a constant
define i,j as an integer variable

''create the window
create window
let title(window) = 
  "Example 3: Using cameras to show different views of the same scene"
call display(window) ''display the window now so that we know the
''size of the canvas!

''create the world
create world
file this world in world_set(window)

''create a light source and point it in the same direction as the camera
create light
call set_forward(light)(0.0, 0.0, -1.0)
file this light in light_set(world)

''create a background node to hold the objects
create background
file this background in node_set(world) ''add to "root" of scene-graph
"create the one model for the tank
create tank_model
call read(tank_model)("tank.3ds", "")
file tank_model in model_set(world)

"create the tank node
create the_tank
let model(the_tank) = tank_model
file the_tank in node_set(world)  'add to "root" of scene-graph
file the_tank in motion_set

"create the "tank" camera.  It will be attached to the tank's 3dnode
"so that it will show the view out of the tank
create tank_camera
call set_perspective(tank_camera)(60.0, 1.0, 0.1, 115.0, 1)
call set_forward(tank_camera)(0.0, 0.0, 1.0)
call set_location(tank_camera)(0.0, 4.0, 0.0)
call set_viewport(tank_camera)(0, 0, canvas_width(window)/2,
        canvas_height(window), 1)
file tank_camera in camera_set(world)
file tank_camera in node_set(the_tank)

"create the "lot" camera, place it along positive z axis but
"point it down the negative z direction
create lot_camera
call set_perspective(lot_camera)(60.0, 1.0, 0.1, 115.0, 1)
call set_forward(lot_camera)(0.0, 0.0, -1.0)
call set_location(lot_camera)(0.0, 4.0, 60.0)
call set_viewport(lot_camera)(canvas_width(window)/2, 0,
        canvas_width(window)/2,
        canvas_height(window), 1)
file this lot_camera in camera_set(world)

"make the lot_camera "track" the tank
let tracked_node(lot_camera) = the_tank

"create the model for the cone
create cone_model
call read(cone_model)("cone.3ds", "")
file this cone_model in model_set(world)

"create many cones and space them out
reserve cones as _num_cols_cones by _num_rows_cones
for i = 1 to _num_rows_cones
    for j = 1 to _num_cols_cones
        do
            create cones(i,j)  "cones share the same model
            call set_location(cones(i,j))(i * 10 - 55.0, 0.0, j * 10 -55.0)
            file this cones(i,j) in node_set(background)  "add node to scene-graph
        loop
activate a movement(the_tank) now  "our process method
activate a animate(window)(10000) now  "tell window to show animation
let timescale.v = 10  "10/100 real seconds per unit of time
start simulation

end
Figure 20: Example 31, tank program updated to include two viewports and tracking
5.4. Lighting up a scene-graph

As all objects in the scene-graph as smooth-shaded, light source(s) are needed to apply the shading. A light source can be added to the 3dworld by creating one or more instances of a 3dlight object. The 3dlight class is derived from 3dnode and can reside in a scene-graph much the same way that a 3dcamera can. Lights are positioned and oriented using methods inherited from 3dnode.

5.4.1 Determining the “variety” of lighting

In addition, the 3dlight class adds a variety attribute which defines the characteristic nature of the light source. The three varieties are _directional, _positional, and _spot:

<table>
<thead>
<tr>
<th>Value for variety</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3dlight’_directional</td>
<td>Light has no fixed position, but emanates uniformly from a single direction. The set_forward method can be used to set the direction vector. The set_location method has no effect. This light is useful for mimicking sunlight, or light from a far away source. This is the default variety of 3dlight.</td>
</tr>
<tr>
<td>3dlight’_positional</td>
<td>Light emanates uniformly in all directions from a single source. The set_location method can be used to set the location of the source.</td>
</tr>
<tr>
<td>3dlight’_spot</td>
<td>This variety has both direction and positional characteristics. It is shown as a cone of light emanating from a fixed point (set_location method) and traveling in a certain direction (set_forward method) The spot_cutoff attribute is used to set the angle (in degrees). See Figure 21.</td>
</tr>
</tbody>
</table>

Figure 21: The spot_cutoff attribute

A good example of the use of “positional” lighting would be the SIMSCRIPT III “parking lot” demo. This program sets up a parking lot simulation at night. Lamp-posts are positioned at strategic locations around the parking lot. A 3dlight object is located at the top of each lamp-post. The variety attributes are assigned the 3dlight’_positional
The cars are illuminated from different angles and from different light sources as they move through the lot, which creates a realistic effect (see Figure 22).

5.4.2 Ambient, Diffuse and Specular light

When light strikes the surface of an object, it is reflected based upon the properties of the surface material, and on the normal vector of the surface in relation to the light’s direction. Surface normals can be specified along with the geometry of a shape and this is described later. For now we will assume that the program will be using a 3dmodel that specifies the correct normal vectors for each surface. For each 3dlight we can define how it interacts with the surface that is reflecting it. Relative amounts of ambient, diffuse and specular light can be specified by assigning the ambient_color, diffuse_color and specular_color attributes. These types of light are described below:

Ambient
This type of light is non-directional and non-positional (location and orientation is ignored). Ambient light will illuminate all surfaces ignoring normal vectors. The ambient term is used simply to keep shadows from turning pitch black. The relative intensity of ambient light can be set by assigning the \texttt{ambient\_color} attribute to an rgb value returned from the \texttt{gui.m:color'rgb} class method. If less ambient light is required, darker colors should be used. For example, setting \texttt{ambient\_color} to \texttt{color'rgb(0.25, 0.25, 0.25)} will provide a $1/4^\text{th}$ intensity of ambient light.

Diffuse

Diffuse light is reflected from a surface in all directions. The amount of reflected light is determined by the “diffuse” color of surface material (see \texttt{3dmaterial}). Rough surfaces should have a relatively bright \texttt{diffuse\_color} attribute. \textit{Both} diffuse and specular light allows objects to be shaded based on surface normal vectors. Surface elements with normals pointing at the light source will be illuminated while surfaces with normals orthogonal to the light source’s direction will not. The \texttt{diffuse\_color} attribute determines the color and intensity of diffuse light.

Specular

Specular light is reflected from a surface in mostly one direction. Shiny surfaces reflect more of the specular light than rough surfaces. The amount of reflected light is determined by the “specular” color of surface material (see \texttt{3dmaterial}). Surface elements with normals pointing at the light source will be illuminated while surfaces with normals orthogonal to the light source’s direction will not. The \texttt{specular\_color} attribute determines the color and intensity of this light.

All color related attributes defined in the \texttt{3d.m} subsystem are specified in terms of an rgb triple. This is an integer returned from the “color’rgb” class method which is found in the \texttt{gui.m} subsystem. The \texttt{color’rgb} class function takes three double parameters as the percentage of RED, GREEN, and BLUE in the color (all in the range $[0,1]$). Predefined colors are defined in the color class also. For example, to set specular color to a particular shade of yellow:

\begin{verbatim}
Let specular_color(my_light) = color'rgb(0.9, 0.9, 0.0)
\end{verbatim}

Or to set specular color to red:

\begin{verbatim}
Let specular_color(my_light) = color'_red
\end{verbatim}

The witnesses effect of the light striking a \textit{3d} object and reflecting into the camera depends on the ambient, diffuse and specular colors attributes of not only the \texttt{3dlight} but also of the \texttt{3dmaterial} (which represents the surface properties of the object, and is described later). If specular (shiny) effects are to be seen, usually both the material and light must have a high degree of specular color. Figure 23 shows how specular light can affect the appearance of a surface. The scene is illuminated with a directional light from above (I.e. \texttt{set\_forward(0.0, -1.0, 0.0)} and with the \texttt{specular\_color} attribute of the \texttt{3dlight}
set to color’_white. As far as the toruses go, the ambient and diffuse color for all is set to a shade of purple or color’rgb(0.8, 0.4, 0.7). However, the specular_color attribute for toruses on the right is greater than those on the left.

Figure 23: Objects on the left have more specular color and appear shiny.

5.4.3 Setting up the lighting using the 3dlight

To summarize, setting up lighting involved the following set of steps:

a. Create instance(s) of a 3dlight object.
b. Optionally set the location the direction of the light using the set_location and set_forward methods inherited from 3dnode.
c. Set the type of light by assigning the variety attribute. (_positional, _directional, _spot).
d. Assign the ambient_color, diffuse_color, and specular_color attributes in accordance with the desired properties.

e. File the 3dlight object instance into the light_set owned by the 3dworld that will contain the light.

f. Optionally file the light into a node_set if it is to be attached to an object in the scene-graph.

In examples 30 and 31 above, a single directional light source is used. The light is pointed away from the viewer—along the negative Z axis by calling the set_forward method. (If the light were to be directed along the positive Z axis, the objects in the foreground would be “back-lit” and difficult to see). The default ambient, diffuse and specular color attributes are used.

```
'create a light source and point it in the same direction as the camera
create light
call set_forward(light)(0.0, 0.0, -1.0)
file this light in light_set(world)
```

We can make the lighting more interesting in this example by creating a “spot” variety light and automatically pointing it at the tank as it moves around. For the spot light, a new class called tank_spotter is derived from 3dlight. To achieve the automated tracking, the motion method is overridden. (This method is described later – it is called automatically as simulation time is updated). The code to declare the subclass of 3dlight is:

```
begin class tank_spotter
  every tank_spotter is a 3dlight
  and overrides the motion
end
```

The implementation code for motion is:

```
method tank_spotter'motion(dt)
  'point the light at the tank
  call aim(location_x(the_tank), location_y(the_tank), location_z(the_tank))
end
```

To initialize the light its variety attribute is assigned to spot and we must provide its location. Also, since a spot light is being used, we should define the spot_cutoff attribute—it will be set to 15 degrees. Lastly, in order for the light’s motion method to be called automatically, it must be filed into the motion_set (described later).

```
'create the tank_spotter light and position it near the camera
create light
let variety(light) = 3dlight'_spot
let spot_cutoff(light) = 15  ''degrees-defines the light cone
call set_location(light)(0.0, 6.0, 60.0)
file this light in light_set(world)
file this light in motion_set
```
Figure 24: Using a spot light to track the tank.
5.5 Loading graphics files via Models

3d dimensional objects are easily maneuvered using the SIMSCRIPT 3d graphics functionality, but creating three dimensional cars, tanks, and airplanes by program code can be a long process. For this reason, it is much easier to load predefined objects from a graphics file format such as Autodesk’s “3ds”. These 3d graphics can be created using a 3d “point and click” editor, or can be purchased online. In any case, SIMSCRIPT 3d graphics supports the loading of two well known 3d graphics file formats – 3ds and dxf.

5.5.1 Reading in the 3dmodel from a file

The 3d surfaces and geometry shown in the 3dworld can be defined in one of two ways. The 3dgraphic and its subclasses allow the application to specify the geometry and materials and runtime. The 3dmodel class allows the surfaces and materials to be loaded from a file. Basically, a single instance of a 3dmodel is created for each separate 3d file. The read method loads the contents of the file creating surfaces and materials, which are saved in memory.

```plaintext
define read as a method given
  1 text argument,  '''file name including .3ds, .dxf, .sg2 extension
  1 text argument     '''name of model in the file (.sg2 files only)
```

The first argument specifies the name of the file. Currently, the file must be in either autodesk 3dStudio or “.3ds” format (the extension is required), AutoCAD dxf format, or SIMGRAPHICS II “.sg2” format. For .sg2 files, the name of the model within the file is provided in the second argument. Also, each 3dmodel instance must be filed into a model_set owned by the 3dworld, which will show the model.

From Example 29, we load the model of the 3d car from the file named “ford.3ds” as follows:

```plaintext
define model as a 3dmodel reference variable
. . .
create model
call read(model)("ford.3ds", "")
file this model in model_set(world)
```

5.5.2 Linking a 3dmodel instance to a 3dnode instance

The 3dmodel class is not derived from 3dnode and therefore cannot be shown in a window directly. The 3dmodel instance can be assigned to the model attribute of a 3dnode. This will provide a link from the image of the 3dmodel to the 3dnode. Many instances of 3dnode can reference the same 3dmodel instance. This scheme allows a single model to be drawn in different locations and orientations in the world. (See Figure 25).
5.5.3 Copying the scene-graph of a 3dmodel

In some cases, the application may require individual images of a 3dmodel to appear differently. In this case the same model cannot be shared—each image is different and requires its own separate scene-graph. For example, if many “tanks” were to be displayed, each turret will have a different orientation. In this case, instead of assigning the *model* attribute, the *3dnode* `load` method would be called. This method will make copies of each sub-component and place the nodes into the appropriate node_set.

Define tank1, tank2, as tank reference variables
Define the_model as a 3dmodel reference variable

Create tank1, tank2, the_model
File the_model in model_set(the_world)
File tank1 in node_set(the_world)
File tank2 in node_set(the_world)
Call read(the_model)("tank.3ds", ")
Call load(tank1)(the_model)
Call load(tank2)(the_model)
5.5.4 Locating named sub-components

Models may be designed in the 3d editor to have well defined components. In the previous example, the ‘tank’ model has a component that was named “turret1” in the editor for the purpose of being rotated independently with respect to the tank. The turret sub-component may in turn have a ‘gun’ sub-component. The application may need access to these sub-components at runtime (i.e. rotate the turret, move the gun in and out when it fires, etc). If sub-components like this are defined in the graphics editor, they will be preserved when model is loaded in the application. The \textit{node\_set} owned by the \textit{3dmodel} will contain these components. Both the \textit{3dnode} and \textit{3dmodel} classes define a \textit{find} method that can be called to perform a depth-first search for a sub-component, provided that the component has been given a name in the graphics editor.

Let \text{turret(tank1)} = \text{find(tank1)}("turret1")
Let \text{turret(tank2)} = \text{find(tank2)}("turret1")

5.5.5 Deriving from sub-components

Suppose the application wanted to represent the turret sub-component of a tank using an object \textit{derived} from \textit{3dnode} (instead of the \textit{3dnode} base class). In this case the \textit{3dmodel} would be sub-classed and its \textit{create\_component} method overridden.

\begin{verbatim}
define create\_component as a "virtual" 3dnode reference method given
  1 text argument,       "name of the component
  1 text argument        "name of required class

  "This method is called by the system during the execution of the
  ""read" method to create a new sub-node component.
  "The name given to the component in the model file
  "is provided in argument 1. By default this method will create a
  "instance of the class named by argument 2, but can be overridden to
  "create a sub-class of arg 2.
\end{verbatim}
Create_component is called automatically for each separate component that is created at the time a 3dmodel is read. Overriding this method allows you to create and return the correct sub-class that should represent the component. In the above example, create_component would be implemented to create and return a “turret” object if the first argument specified the name assigned to the turret component in the graphics editor. See below.

```plaintext
method tank_model'create_component(component_name, class_name)
    define my_turret as a turret reference variable
    if component_name = "turret1" and class_name = "3dnode"
        create a turret called my_turret
        return with my_turret
    otherwise
        return with 3dmodel'create_component(component_name, class_name)
end
```

When the 3dnode’load method is used, it will make copies of all components in the model. Individual attributes are copied by an internal call to the copy_attributes method. Therefore, if a component is subclassed as above, and the load method is to be used, the sub-class must override this method if it defines attributes that need to be copied with the rest of the components. In our tank example, suppose the “turret” class defines attributes “azimuth” and “attitude” that are initializes in the model. We want these values to be propagated when turret is copied (via the load method).

```
Begin class turret
    Every turret is a 3dnode and has
    A azimuth,
    An attitude, and
    Overrides the copy_attributes
    Define azimuth, attitude as double variables
End
```

The copy_attributes method’s implementation would look like this:

```plaintext
method turret'copy_attributes(node)
    define p as a pointer variable
    define original_turret as a turret reference variable
    let p = node  "necessary due to original prototyping
    let original_turret = p
    let azimuth = azimuth(original_turret)
    let attitude = attitude(original_turret)
    call 3dnode'copy_attributes(node)
end
```

In the next example, a tank model is used that has its turret represented by a subcomponent called “turret1”. Three sub-classes are defined in the preamble: tank_model, a subclass of 3dmodel, tank a subclass of 3dnode, and turret another subclass of 3dnode. The tank_model class will override create_components to create and instance of a turret object when “turret1” is passed as the component name.
Both the tank and turret subclasses will define a process method called *movement*. Each process method will be implemented in a unique fashion to allow simultaneous but different motion.

```
'Example 32: Acessing the subcomponents of a 3d tank
Preamble including the 3d.m, 3dshapes.m subsystem
begin class tank_model
    every tank_model is a 3dmodel
    and overrides the create_component
end

begin class tank
    every tank is a 3dnode and has
    a speed,
    a spin_rate,
    a forward process method,
    a spin process method,
    a movement process method,
    overrides the motion
    define speed, spin_rate as double variables
    define spin, forward as process methods
    given 1 double argument
end

begin class turret
    every turret is a 3dnode and has
    a spin_rate,
    a spin process method,
    a movement process method,
    overrides the motion
    define spin_rate as a double variable
    define spin as a process method given 1 double argument
end

define window as a 3dwindow reference variable
end

method tank_model'create_component(component_name, class_name)
    define my_turret as a turret reference variable
    if component_name = "turret1" and class_name = "3dnode"
        create a turret called my_turret
        return with my_turret
    otherwise
        return with 3dmodel'create_component(component_name, class_name)
end

process method tank'forward(distance)
    define _forward_speed=2 as a constants
    let speed = _forward_speed
    wait abs.f(distance) / speed units
    let speed = 0.0
end

process method tank'spin(angle)
    define _spin_speed = 20 as a constants
    let spin_rate = _spin_speed * sign.f(angle)
    wait abs.f(angle) / _spin_speed units
    let spin_rate = 0
end
```
process method tank'movement
    call forward(40)
call spin(180)
call forward(40)
call spin(90)
call forward(40)
call spin(90)
end

process method turret'spin(angle)
define _spin_speed = 10 as a constant
    let spin_rate = _spin_speed * sign.f(angle)
    wait abs.f(angle) / _spin_speed units
    let spin_rate = 0
end

process method turret'movement
    call spin(45)
    wait 2.0 units
    call spin(-90)
    wait 1.0 units
    call spin(180)
    call spin(-30)
    wait 1.0 units
    call spin(-40)
end

'our motion method will be called as simulation time advances
method tank'motion(dt)
    call 3dnode'motion(dt)
call rotate_y(dt * spin_rate)
call move(0.0, 0.0, dt * speed)
end

method turret'motion(dt)
    call rotate_y(dt * spin_rate)
end

main
    define world as a 3dworld reference variable
    define camera as a 3dcamera reference variable
    define light as a 3dlight reference variable
    define tank_model as a tank_model reference variable
    define the_tank as a tank reference variable
    define the_turret as a turret reference variable
    define p as a pointer variable

    'create the window
create window
let title(window) = "Example 5: Accessing sub-components of a 3dmodel"

    'create the world
create world
file this world in world_set(window)

    'create camera, place it along positive z axis but
    'point it down the negative z direction
create camera
call set_perspective(camera)(90.0, 1.0, 5.0, 115.0, 1)
call set_forward(camera)(0.0, 0.0, -1.0)
call set_location(camera)(0.0, 4.0, 60.0)
file this camera in camera_set(world)
'create a light source and point it in the same direction as the camera
create light
call set_forward(light)(0.0, 0.0, -1.0)
file this light in light_set(world)

'create the one model for the tank
create tank_model
call read(tank_model)("tank.3ds", "")
file tank_model in model_set(world)

'create the tank node
create the_tank
call load(the_tank)(tank_model)   ''copy components of tank to node
let p = find(the_tank)("turret1")  ''locate the "turret1" node
let the_turret = p
file the_tank in node_set(world)  ''add to "root" of scene-graph
file the_tank in motion_set
file the_turret in motion_set

call display(window)   ''bring up the main window and show everything

activate a movement(the_tank) now
activate a movement(the_turret) now       ''our process method
activate a animate(window)(10000) now    ''tell window to show animation
let timescale.v = 50     ''50/100 real seconds per unit of time

start simulation
end

5.5.6 Misc. 3dmodel options

There are a couple more options regarding the 3dmodel that can be set before the model is read from the file. They are _smoothing, and _cache_model. The options are set via a left handed use of the enabled method. For example:

define the_model as a 3dmodel reference variable
...
let enabled(the_model)(3dmodel'_smoothing) = 1   ''turn on smoothing
let enabled(the_model)(3dmodel'_smoothing) = 0   ''turn off smoothing

The _smoothing option will cause normal vectors to be recomputed at the time the model is read from the file. This will give the surfaces in the model a smooth appearance. It is off (0) by default.

The _cache_model option, if on, will cause the runtime library to create an internal “call list” for the model at the time a 3dnode which references it (via the model attribute) is first drawn. For the case of multiple 3dnode objects referencing the same static model, this will improve performance. This option is not relevant if the load method is used to link the 3dnode to the model.

After a model is read, its size can be determined if necessary. The get_bounding_box method can be called to get the dimensions of a model in the x, y and z directions.
define get_bounding_box as a method yielding
3 double arguments, "(xlo,ylo,zlo)
3 double arguments "(xhi,yhi,zhi)
"Computes the smallest 3d box that will enclose the model
"Must be called after "read" to be effective.

Instances of 3dmodel must be filed into the model_set owned by the 3dworld in which the model is to appear. This must be done before the window is displayed. Materials used in the model are automatically filed into the material_set owned by 3dmodel when the 3dmodel 'read' method is called.
5.6 Geometry

Often times parts of a 3d scene cannot be stored in a .3ds or .dxf “3dmodel” file. If it is the case that the geometry is not known exactly until runtime, the programmer must write code required to construct this geometry. SIMSCRIPT III provides several classes that allow this.

The 3dgraphic is the base class for all objects whose geometry is defined by program code, and not by offline models stored in a file. This class is derived from 3dnode and is part of the scene-graph. Since 3dgraphic is derived from 3dnode, instances must be filed into a node set (owned by either the 3dworld or by another 3dnode attached to a 3dworld). The 3d.m and 3dshapes.m subsystems provide several classes derived from 3dgraphic that can be useful for constructing various 3d objects (see below).

<table>
<thead>
<tr>
<th>Class name</th>
<th>Description</th>
<th>Geometry attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3dgraphic</td>
<td>Can be used to draw any type of shape.</td>
<td>Override draw method</td>
</tr>
<tr>
<td></td>
<td></td>
<td>draw_normal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>draw_texture_coordinate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>draw_vertex</td>
</tr>
<tr>
<td>3dfaces</td>
<td>Tessellated surface, (triangles, quads)</td>
<td>points, normals attributes</td>
</tr>
<tr>
<td>3dlines</td>
<td>Line segments in 3d space</td>
<td>points, width attributes</td>
</tr>
<tr>
<td>3dpoints</td>
<td>Single points in 3d space</td>
<td>points, size attributes</td>
</tr>
<tr>
<td>3dtext</td>
<td>2d text in 3d space</td>
<td>string, width, height, font</td>
</tr>
<tr>
<td>3dbox</td>
<td>Found in 3dshapes.m. Cubes,</td>
<td>width, height, depth</td>
</tr>
<tr>
<td>3dcone</td>
<td>Simple cone shape.</td>
<td>length, radius attributes</td>
</tr>
<tr>
<td>3dcylinder</td>
<td>Simple cylinder shape.</td>
<td>length, radius attributes</td>
</tr>
<tr>
<td>3dellipse</td>
<td>2d circle, arc, pie or ellipse in 3d space</td>
<td>radius_x, radius_y, start_angle, stop_angle</td>
</tr>
<tr>
<td>3drectangle</td>
<td>2d rectangle in 3d space</td>
<td>height, width</td>
</tr>
<tr>
<td>3dsphere</td>
<td>Simple sphere shape</td>
<td>radius</td>
</tr>
</tbody>
</table>

5.6.1 Using the 3dgraphic class

Occasionally, there are cases where the exact shape is not known until runtime. For example, a simulation shown a 3d terrain landscape may not know the geographical area to display until runtime. Even then, the coordinates of the “hills and valleys” may reside in a data file which must be decoded and the vertex information extracted or computed. We can handle this case using the 3dgraphic class.

Geometry and surface materials are specified by sub-classing 3dgraphic and overriding the draw method. The draw method is called automatically when the system needs to have the 3dgraphic object rendered. (Calling the 3dwindow’display method will invoke this method. Draw may also be called as a result of event handling). The draw method
should be programmed to make calls to `begin_drawing` and `end_drawing` class methods to define each surface. After the call to `begin_drawing`, calls to class methods such as `draw_normal`, `draw_texture_coordinate`, and `draw_vertex` allow geometry to be specified for the surface. A call to `end_drawing` will mark the end of the geometry description started by `begin_drawing`. We will refer to this as a “drawing block”.

There can be multiple drawing blocks defined in a single `draw` method. It is also possible to sub-class the `3dgraphic` and override the `draw` method to “add” geometry. Nested drawing blocks are not allowed and will be flagged as a runtime error.

There are several formats of geometry that can be specified in a drawing block. A “format” argument is passed to the `begin_drawing` method. The `3dgraphic` class provides to following predefined constants that can be passed to `begin_drawing`. (The formats are described in detail later).

<table>
<thead>
<tr>
<th>Constant</th>
<th>Format of Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td>_points</td>
<td>Draw dots in 3d space. Each call to <code>draw_vertex</code> adds a dot.</td>
</tr>
<tr>
<td>_lines</td>
<td>Each 2 successive vertices defines a line segment.</td>
</tr>
<tr>
<td>_line_loop</td>
<td>Each vertex is connected to the next with the first connected to last.</td>
</tr>
<tr>
<td>_line_strip</td>
<td>Each vertex is connected to the next.</td>
</tr>
<tr>
<td>_triangles</td>
<td>Each successive group of 3 vertices defines a triangle</td>
</tr>
<tr>
<td>_triangle_strip</td>
<td>For each index n &gt; 2 a triangle is defined by vertex at n-2, n-1 and n in counter-clockwise order. Sometimes this is called the “triangular mesh” or just “mesh”.</td>
</tr>
<tr>
<td>_triangle_fan</td>
<td>For each index n &gt; 2 a triangle is defined by vertex at n, n-1 and 1.</td>
</tr>
<tr>
<td>_quads</td>
<td>Each successive group of 4 vertices defines a quadrilateral face.</td>
</tr>
<tr>
<td>_quad_strip</td>
<td>For each 2 indices n, n-1, a new quadrilateral is composed of vertices at n-3, n-1, n, n-2 in counter-clockwise order</td>
</tr>
<tr>
<td>_polygon</td>
<td>The outline of the polygon is defined by vertices. Again, the vertices for front facing polygons should be arranged in a counter-clockwise order.</td>
</tr>
</tbody>
</table>

### 5.6.2 Surface geometry

The surface of a shape in 3d is composed of small by planar faces that are interconnected. Round shapes that have smaller but more numerous faces will appear more detailed, but could take longer to render. Calls to the `draw_vertex` are placed inside a drawing block with the (x,y,z) values passed as arguments. Each call adds a new coordinate to the shape.

A “normal vector” is perpendicular to the surface that is being drawn. If a normal vector is known, it can be specified by calling `draw_normal`. The normal vector specified will apply to subsequent vertices defined by `draw_vertex`. 
To construct a 2d surface, one of the following predefined constants must be passed to \textit{begin\_drawing}: \_triangles, \_triangle\_strip, \_triangle\_fan, \_quads, or \_polygon. Using the \_triangles format will draw a separate triangle for each successive group of three vertices. As a rule of thumb, the vertices should be given in a counter-clockwise ordering if the normal vector points outward.

\textbf{Figure 28: \_triangles format.}

The \_triangle\_strip format is sometime referred to as a “triangular mesh” or just “mesh”. It can be used to construct surfaces composed of interconnected triangles. Each call to \textit{draw\_vertex} will add a new triangle to the shape using the two previous coordinates given by \textit{draw\_vertex}. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure28.png}
\caption{Figure 28: \_triangles format.}
\end{figure}
When using the _triangle_fan format, draw_vertex will add a new triangle using the first vertex and the previous vertex. This format can be useful for drawing circular shapes.

Surfaces composed of quads or 4-point planar regions can be defined. When using the _quads format, each group of 4 vertices defines a new quadrilateral face.
Much like the “triangle strip” a quad strip can also be constructed. Each two successive vertices will add a new quad using the two previous vertices for the face.

A simple polygon can also be created. The polygon must not have any concave portions or it may not be rendered correctly. The outline of the polygon is defined by successive calls to `draw_vertex`. Again, the points for front facing polygons should be arranged in a counter-clockwise ordering.
In the following example we will develop a new shape to represent a torus. The code to construct this doughnut shaped object will be entered into the overridden `draw` method that is inherited from the `3dgraphic` class. The actual code to draw the torus will, in a loop, construct successive cylindrical rings that run in a circle along the shape of the torus. A “begin_drawing / end_drawing” block will construct each ring as a _quad_strip_. A normal vector is defined for each vertex. This is computed as the vector originating from the center of the current ring to the current vertex.

Figure 34: A torus composed of rings made of quad strips
Sub-classing the 3dgraphic class will allow us to add additional attributes that parameterize the torus, such as the major and minor radii, and the number of facets. The complete code is shown below:

Here's Example 33 a 3d torus
preamble for the toruses system including the 3d.m subsystems
begin class torus
every torus is a 3dgraphic and has
a material,
a numMajor,
a numMinor,
a minorRadius,
a majorRadius, and
overrides the draw

define material as a 3dmaterial reference variable
define numMajor, numMinor as integer variables
define minorRadius, majorRadius as double variables
end
end

this method is called automatically. We will provide the code needed
to draw a torus
method torus'draw
define nx, ny, nz as double variables
define a0, a1, x0, y0, x1, y1 as double variables
define b, c, r, z, majorStep, minorStep as double variables
define i, j as integer variables

let majorStep = 2.0*pi.c / numMajor
let minorStep = 2.0*pi.c / numMinor

call set_material(material)

for i=0 to numMajor-1
do
a0 = i * majorStep
a1 = a0 + majorStep
x0 = cos.f(a0)
y0 = sin.f(a0)
x1 = cos.f(a1)
y1 = sin.f(a1)

call begin_drawing(_quad_strip)

for j=0 to numMinor
do
b = j * minorStep
c = cos.f(b)
r = minorRadius * c + majorRadius
z = minorRadius * sin.f(b)

First point
call 3d.m:vector_normalize(x0*c,y0*c,z/minorRadius) yielding nx,ny,nz
call draw_normal(nx, ny, nz)
call draw_vertex(x0*r, y0*r, z)
call 3d.m:vector_normalize(x1*c,y1*c,z/minorRadius) yielding nx,ny,nz
call draw_normal(nx, ny, nz)
call draw_vertex(x1*r, y1*r, z)
loop
call end_drawing
loop
end

main
define window as a 3dwindow reference variable
define world as a 3dworld reference variable
define camera as a 3dcamera reference variable
define light as a 3dlight reference variable
define torus as a torus reference variable

''create the window
create window
let title(window) = "Example 7: A torus"

''create the world
create world
file this world in world_set(window)

''create a camera 3 units out, pointed at the origin
create camera
call set_perspective(camera)(60.0, 1.0, 0.1, 100.0, 1)
call set_forward(camera)(0.0, 0.0, -1.0)
call set_location(camera)(0.0, 0.0, 3.0)
file this camera in camera_set(world)

''create our torus and set its size and detail parameters
create a torus
let minorRadius(torus) = .3
let majorRadius(torus) = .7
let numMinor(torus) = 37
let numMajor(torus) = 61

''create a "red" material for torus's surface
create a material(torus)
let ambient_color(material(torus)) = color'rgb(0.2, 0.0, 0.0)
let diffuse_color(material(torus)) = color'_red
let specular_color(material(torus)) = color'_white
let shininess(material(torus)) = 1.0
file this material(torus) in material_set(world)

''save the torus in the "world"
file this torus in node_set(world)

''create a light from above, set its color
create light
let diffuse_color(light) = color'rgb(0.8, 0.8, 0.8)
call set_orientation(light)(0.0, -1.0, 0.0, 1.0, 0.0, 0.0)
file this light in light_set(world)

call display(window)  ''show every thing

let timescale.v = 100    ''1 second per 1 unit simulated time
activate a animate(window)(10000) now
start simulation
end
5.6.4 Surface appearance

3dgraphic has class methods that can be called to define the properties of the surface. The `set_color` method can be used to set the color or, if called before `draw_vertex`, can set the color applied to individual vertices.

```
define set_color as a method given
    1 integer argument, ''_front, _back, or _front_and_back
    1 integer argument, ''_ambient, _diffuse, _specular
    1 integer ''color'' argument       ''color (color'rgb(r,g,b))
```

The first argument indicates which side the color should be applied to, and must be one of the constants _front, _back or _front_and_back. The second argument gives the type of lighting, _ambient, _diffuse, or _specular that should reflect the given color (see `3dlight`). The third argument is the color value obtained from the `gui.m:color'rgb` class.

Attributes of the `3dmaterial` object (colors and texture) can be applied to the `3dgraphic` as well. The `set_material` method should be called before `begin_drawing` and takes a `3dmaterial` instance as an argument. This instance must be filed in the `3dworld'material_set` at initialization.

```
define set_material as a method given
    1 3dmaterial reference argument    ''pointer to material
```
The *shininess* of the surface (used for specular lighting) is set by calling the *set_shininess* class method.

```
define set_shininess as a method given
  1 integer argument,     ''_front, _back, or _front_and_back
  1 double argument      ''shininess (0.0 to 1.0)
```

The first argument must be one of the constants _front, _back, or _front_and_back. The second “shininess” argument must be a value between 0.0 and 1.0 (see 3dlight).

By default, both the front and back sides of a surface are visible. In some cases, back or front side of an object is never seen. Faster rendering can be achieved for single sided drawings by defining which side is to be made visible. The *set_visibility* method can be called to accomplish this. The method call must be made before the call to *begin_drawing*.

```
define set_visibility as a method given
  1 integer argument  ''_front, _back or _front_and_back
```

Individual edges for surfaces specified by calls to *draw_vertex* are normally visible. For some edges such as those on the interior of a complicated shape, this visibility can lead to unwanted artifacts when the 3dgraphic is displayed. The *set_edge_visibility* method can be used to hide or show future edges added to the shape by *draw_vertex*.

```
define set_edge_visibility as a method given
  1 integer ''boolean'' argument       ''0=>invisible, 1=>visible
    ''specifies visibility of future edges of a face
```

### 5.6.5 Surface texture mapping

Texture mapping can be applied to surfaces created in a drawing block. The *draw_texture_coordinate* method can be called to map the vertex (specified by *draw_vertex*) to a 2d texture coordinate (texture mapping is explained in more detail later).

The previous example could be modified to wrap a texture over the surface of the torus. The image that will be mapped is that of a soda can label (saved in the file “cocacola.bmp”). The surface will be parameterized to the (s,t) coordinate space of used for texture mapping whose domain is s => [0,1] and t => [0,1]. (0,0 is the lower left corner of the 2d image and (1,1) is the upper right corner.) For each call to *draw_vertex* a corresponding call to *draw_texture_coordinate* is added. Also the *texture_name* attribute of the *3dmaterial* used to set the surface color will be assigned to “cocacola.bmp”.

We will also modify the color of the material used for the torus. The 2d image file will provide the color information, and since we don’t want that color to be modified, _white
will be used for the diffuse color and a dark gray for the ambient color. The new example code is shown below with the modified and new color colored red.

```
''Example 34: a texture mapped 3d torus
preamble for the toruses system including the 3d.m subsystems
begin class torus
  every torus is a 3dgraphic and has
    a material,
    a numMajor,
    a numMinor,
    a minorRadius,
    a majorRadius, and
  overrides the draw

  define material as a 3dmaterial reference variable
  define numMajor, numMinor as integer variables
  define minorRadius, majorRadius as double variables
end
end

''this method is called automatically. We will provide the code needed
'to draw a torus
method torus'draw
  define nx, ny, nz as double variables
  define a0, a1, x0, y0, x1, y1 as double variables
  define b, c, r, z, majorStep, minorStep as double variables
  define i, j as integer variables

  let majorStep = 2.0*pi.c / numMajor
  let minorStep = 2.0*pi.c / numMinor

  call set_material(material)
  for i=0 to numMajor-1
    do
      a0 = i * majorStep
      a1 = a0 + majorStep
      x0 = cos.f(a0)
      y0 = sin.f(a0)
      x1 = cos.f(a1)
      y1 = sin.f(a1)

      call begin_drawing(_quad_strip)
      for j=0 to numMinor
        do
          b = j * minorStep
          c = cos.f(b)
          r = minorRadius * c + majorRadius
          z = minorRadius * sin.f(b)

          ''for texture coordinates, map entire surface of torus to [0,1]
          call 3d.m:vector_normalize(x0*c,y0*c,z/minorRadius) yielding nx,ny,nz
          call draw_texture_coordinate(j / numMinor, i / numMajor)
          call draw_normal(nx, ny, nz)
          call draw_vertex(x0*r, y0*r, z)
          call 3d.m:vector_normalize(x1*c,y1*c,z/minorRadius) yielding nx,ny,nz
          call draw_texture_coordinate(j / numMinor, (i+1) / numMajor)
          call draw_normal(nx, ny, nz)
          call draw_vertex(x1*r, y1*r, z)
      loop
```
call end_drawing
loop
end

main

define window as a 3dwindow reference variable
define world as a 3dworld reference variable
define camera as a 3dcamera reference variable
define light as a 3dlight reference variable
define torus as a torus reference variable

''create the window
create window
let title(window) = "Example 8: A soda can torus"

''create the world
create world
file this world in world_set(window)

''create a camera 3 units out, pointed at the origin
create camera
call set_perspective(camera)(60.0, 1.0, 0.1, 100.0, 1)
call set_forward(camera)(0.0, 0.0, -1.0)
call set_location(camera)(0.0, 0.0, 3.0)
file this camera in camera_set(world)

''create our torus and set its size and detail parameters
create a torus
let minorRadius(torus) = .3
let majorRadius(torus) = .7
let numMinor(torus) = 37
let numMajor(torus) = 61

''create a "red" material for torus's surface
create a material(torus)
let ambient_color(material(torus)) = color'rgb(0.2, 0.2, 0.2)
let diffuse_color(material(torus)) = color'_white
let specular_color(material(torus)) = color'_white
let shininess(material(torus)) = 1.0
let texture_name(material(torus)) = "cocacola.bmp"
file this material(torus) in material_set(world)

''save the torus in the "world"
file this torus in node_set(world)

''create a light from above, set its color
create light
let diffuse_color(light) = color'rgb(0.8, 0.8, 0.8)
call set_orientation(light)(0.0, -1.0, 0.0, 1.0, 0.0, 0.0)
file this light in light_set(world)

call display(window)  ''show every thing

let timescale.v = 100  ''1 second per 1 unit simulated time
activate a animate(window)(10000) now
start simulation
end
Figure 36: Example 34 -- A texture mapped torus.

5.6.6 Drawing points and lines

The *begin_drawing* method will accept line and point formats as well as surface formats. Lines are 1-dimensional but are constructed with 3d coordinates. These objects are useful as annotations. In Figure 37, a line loop is used to delineate the orbits of the planets.
Figure 37: Using the _line_loop format to mark planetary orbits

Line related formats constants that can be passed to the begin_drawing method include _lines, _line_strip, and _line_loop. The _lines format is used for drawing multiple line segments. Successive calls to the draw_vertex method mark the endpoints of the segments.

Figure 38: The _lines format
The \textit{line_strip} format is used to draw a polyline. In this format, a new segment is added each time \texttt{draw_vertex} is called.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{line_strip.png}
\caption{The \textit{line_strip} format}
\end{figure}

The \textit{line_loop} format is like the \textit{line_strip} format, except that the last point is automatically connected to the first.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{line_loop.png}
\caption{The \textit{line_loop} format}
\end{figure}

Multiple widths and dash styles are supported. When lines are being drawn, the \texttt{set_pen_size} and \texttt{set_pen_pattern} class methods can be called to set the line width and pattern. These methods must be called before \texttt{begin_drawing}. The current pen size is always in pixels. The \texttt{pattern} is one of the following constants: \texttt{solid}, \texttt{long_dash}, \texttt{dotted}, \texttt{dash_dotted}, \texttt{medium_dash}, \texttt{dash_dot_dotted}, \texttt{short_dash}, \texttt{alternate}. 


5.6.7: Drawing points

“points” are essentially 0-dimentional objects but are also located in 3 dimensions. A point is represented by a visible dot. If the _points format is passed to begin_drawing, each call to draw_vertex will add a “dot” to the scene. The size of these dots in pixels can be set by calling the set_pen_size class method.

For example, suppose we want to draw a “spiral” shape. The preamble would define a subclass of 3dgraphic called “spiral” and override the draw method. The spiral shape will be drawn by moving in a circle with a changing radius. In the code to draw the spiral we place calls to the 3dgraphic class method draw_vertex in between calls to begin_drawing and end_drawing.

```plaintext
''Example 35 a spiral
preamble including the gui.m, 3d.m subsystems
begin class spiral
    every spiral is a 3dgraphic and
    overrides the draw
end
end
```
method spiral'draw
    call set_pen_size(6)

    call begin_drawing(3dgraphic'_points)

    define i, num_points as integer variable

    ''create geometry for spiral
    let num_points = (2.0 * pi.c * 3.0) / 0.1

    ''create spiral points
    for i = 1 to num_points
        do
            call set_color(_front_and_back, _ambient,
                color'rgb(1.0, abs.f(sin.f(0.1 * i)), abs.f(cos.f(0.1 * i))))
            call draw_vertex(0.5 * sin.f(0.1 * i), 0.005 * i, 0.5 * cos.f(0.1 * i))
        loop

    call end_drawing
end

main
    define window as a 3dwindow reference variable
    define world as a 3dworld reference variable
    define spiral_graphic as a spiral reference variable
    define camera as a 3dcamera reference variable

    ''create the window
    create window
    let title(window) = "Example 9: A spiral of points"

    ''create the world
    create world
    let ambient_color(world) = Color'_white   ''needed: no light sources
    file this world in world_set(window)

    ''create the spiral and save it in the scene
    create spiral_graphic
    file this spiral_graphic in node_set(world)

    ''move the spiral
    call set_location(spiral_graphic)(0.0, -0.5, 0.0)

    ''create a camera to show the scene (use default attributes)
    create camera
    call set_location(camera)(0.0, 0.75, 1.5)
    file this camera in camera_set(world)
    call aim(camera)(0.0,0.0,0.0)

    ''show the window
    call display(window)

    while visible(window) <> 0
        call handle.3devents.r
    end
5.6.8 Text

There are many uses for graphic text in a simulation. For example, state information may need to be displayed for an object being simulated. Or maybe the object needs to be “tagged” with a unique text string so that it can be identified. Of course it’s possible that text is “part of” the object being simulated (like a traffic sign). SIMSCRIPT III 3d graphics allows both scalable and non-scalable text to be displayed in a 3dworld. The class used for text display is called 3dtext and is derived from the 3dgraphic class.

The 3dtext class can be used to show a simple 1-line or multi-line text string within the scene-graph. Currently, only predefined fonts are available as class attribute pointers that can be assigned to the font attribute. These fonts are automatically initialized before the SIMSCRIPT program is run. The choice of font affects not only the appearance of the text but also its behavior. The application may require the text to act like part of the scene, in other words to obey the same rules as any other geometric shape with regard to
its transformation from world to canvas (like the text on a stop sign). In this case a vector font should be chosen. However, if text is used for “tagging” or as a simple message, we may not want it to appear larger or smaller as it moves closer or farther from the camera. In this case a raster font is used. To add some detail:

5.6.8.1 Vector fonts
A vector font is rendered by drawing a series of line segments in 3 dimensions. Vector text follows the same rules as do other 3d graphic shapes with regard to location and orientation. The advantage of using this type of text is that it is “part of” the object, for example the text on a road sign, or the monogram on the side of an airplane. The text will increase in size as the camera moves closer to it. The disadvantage is that the text is usually composed of thin lines and may not look good when it is sized big.

The following vector fonts are available:

- 3dtext'stroke_font - Variable width vector font
- 3dtext'stroke_mono_font - Fixed width vector font

The size of vector text is controlled by its width and height attributes. These attributes function the same as the width and height for the 3drectangle class do. The height defines the maximum height including descenders and the width applies to the whole text string.

5.6.8.2 Bitmapped fonts
Characters in a bitmapped or “raster” based font are basically small 2d bitmap images that are copied to the screen when the text is rendered. Text drawn using these types of fonts will always appear right side up regardless of how the 3dtext object (or its owner node) is oriented. However, the 3dnode 'location properties is still utilized. In other words, the text can be positioned by calling the set_location method. Bitmapped text will appear the same size regardless of its distance from the camera. If a larger or smaller text size is needed, a different font must be assigned to the font attribute.

- 3dtext'9_by_15_font - Fixed width bitmap font
- 3dtext'8_by_13_font - Fixed width bitmap font
- 3dtext'times_roman_10_font - Variable width bitmap font
- 3dtext'times_roman_24_font - Variable width bitmap font
- 3dtext'helvetica_10_font - Variable width bitmap font
- 3dtext'helvetica_12_font - Variable width bitmap font
- 3dtext'helvetica_18_font - Variable width bitmap font
For bitmapped fonts, the `align_horiz`, `align_vert` attribute allows the text to be centered, or left/right, top/bottom justified. The following constants can be assigned to the `alignment` attribute:

```plaintext
define _left_justified=0, _centered, _right_justified as constants
  ''for the "aligh_horiz" attribute

define _bottom=0, _middle, _top, _bottom_cell, _top_cell as constants
  ''for the "align_vert" attribute
```
5.6.9 Updating the graphic

By default the `draw` method is only called once by the system when the `3dgraphic` first appears. If the geometry or surface has changed, the `update_drawing` method must be called to indicate that the `3dgraphic` is “old” and that the `draw` method should be invoked (the `draw` method should never be called directly).

```plaintext
define update_drawing as a method given
   1 integer argument    "always, once, never
   _never
```

Figure 44: Text alignment

Figure 45: 3dtext alignment reference points
Draw will not be called. Graphic will effectively be hidden from view.

Draw called the next time the window is refreshed. The image of the 3dgraphic will be “cached” for future refreshes of the screen. The update_drawing method must be called for draw to be invoked again. (This use is most common).

Draw will be called each time the canvas is refreshed. This is useful for objects with constantly changing geometry.

5.6.8 Classes with retained geometry

There are several classes derived from 3dgraphic that are provided in the 3d.m and 3dshapes.m subsystems. When implementing objects derived from these classes, the draw method is NOT overridden. Instead, these classes provide attributes and methods for specifying the shape of the graphic. These classes are useful when the graphic object is static. In other words, the geometry and the number of points/normals does not change during the simulation. In addition, the 3dfaces, 3dlines, and 3dpoints classes support indexed geometry, which may improve performance. These classes are shown below and documented more thoroughly in the subsequent reference section.

<table>
<thead>
<tr>
<th>Class name</th>
<th>Subsystem</th>
<th>Description</th>
<th>Geometry attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3dfaces</td>
<td>3d.m</td>
<td>Tessellated surface, (triangles, quads)</td>
<td>points, normals, attributes</td>
</tr>
<tr>
<td>3dlines</td>
<td>3d.m</td>
<td>Line segments in 3d space</td>
<td>points, width, attributes</td>
</tr>
<tr>
<td>3dpoints</td>
<td>3d.m</td>
<td>Single points in 3d space</td>
<td>points, size, attributes</td>
</tr>
<tr>
<td>3dtext</td>
<td>3d.m</td>
<td>2d text in 3d space</td>
<td>string, width, height, font</td>
</tr>
<tr>
<td>3dbox</td>
<td>3dshapes.m</td>
<td>Cubes, rectangular boxes.</td>
<td>width, height, depth</td>
</tr>
<tr>
<td>3dcone</td>
<td>3dshapes.m</td>
<td>Simple cone shape.</td>
<td>length, radius, attributes</td>
</tr>
<tr>
<td>3dcylinder</td>
<td>3dshapes.m</td>
<td>Simple cylinder shape.</td>
<td>length, radius, attributes</td>
</tr>
<tr>
<td>3dellipse</td>
<td>3dshapes.m</td>
<td>2d circle, arc, pie or ellipse in 3d space</td>
<td>radius_x, radius_y, start_angle, stop_angle</td>
</tr>
<tr>
<td>3drectangle</td>
<td>3dshapes.m</td>
<td>2d rectangle in 3d space</td>
<td>height, width</td>
</tr>
<tr>
<td>3dsphere</td>
<td>3dshapes.m</td>
<td>Simple sphere shape</td>
<td>radius</td>
</tr>
</tbody>
</table>

5.7 Surfaces, Textures and Materials
Surface attributes are important for displaying 3d objects. Accurate specification of the shininess, color and texture of a surface are important for realistic images. In the SIMSCRIPT 3d graphics the 3dmaterial object defines the "skin" of the object being drawn. It is not derived from 3dnode and therefore does not belong in a scene-graph, but it is instead referenced by a 3dnode instance via the 3dnode'material attribute. Not all classes derived from 3dnode make use of the material attribute, however this attribute can be assigned to a parent node – in which case all children will utilize this material.

5.7.1 Setting the color and shininess of a surface

Colors in SIMSCRIPT 3d graphics are specified in an RGB (red-green-blue) triple with element values ranging from 0 to 1. The “rgb” method of the color class in gui.m is used to encode color from RGB values into an integer. For example to obtain a dark green color:

Define dark_green as an integer variable
Let dark_green = color'rgb(0.0, 0.5, 0.0)

A 3dmaterial can be used to define the reflectivity of a surface with respect to diffuse, ambient, and specular colored light. The diffuse_color, ambient_color, and specular_color attributes are used for this purpose.

For specular light the shininess attribute relates to the “specular exponent” of the surface. This value must range from 0 to 1. Higher values lead to smaller, sharper highlights, whereas lower values result in large and soft highlights. If the surface is to be shiny (i.e. metallic in nature) both the specular_color and shininess attributes should be large in value.

In Figure 46, the ambient, diffuse and specular color for the red paint on the car shown in Example 29 is displayed. The “diffuse” color is a dark red with the RGB triple set to (0.56, 0.02, 0.02). The surface will reflect this color in all directions. The ambient color is set to (0.02, 0.0, 0.0). The unlit portions of the car will reflect this (very dark red) color. The specular color is set to (1.0, 1.0, 1.0). This means that the shiny portions of the car will be white.
5.7.2 Texture mapping and raster images

Texture mapping is also supported through the `3dmaterial` class. Texture mapping allows a 2d pixel image (such as a windows .BMP file) to be plastered onto a 3d surface. For larger surfaces, the texture is repeated along the surface much like tiles on a floor. A large brick wall could be simulated using the 2d image of only a few bricks. Of course if a texture is to replicated over a surface the edges of the image must be drawn so that the left edge will mate properly with the right edge.
Figure 47: Top-original texture, Bottom seamless replication of 6 “tiles”

The surface does not need to be flat. In Figure 48, the surface of a sphere is texture mapped to look like the planet Jupiter.
It is possible and often desirable to store each unique face of an object into a single texture bitmap file. The image below shows the features of a car with the top, left, right, back and front sides all contained in the same .bmp file. This will allow a single 3dmaterial instance to represent the surface properties of an entire car.
The mapping of 3d geometry to 2d points in the image file is done using texture coordinates. Basically, when `draw_texture_coordinate` is called before `draw_vertex`, the 2d texture coordinate is mapped or “attached” to the 3d vertex. The result is a smooth texturing of the 2d image over the surface. The `texture_name` attribute of 3dmaterial can be assigned the name of the file containing the image. Currently, only TARGA graphic (.tga) files and Window Bitmap (.bmp) files are supported. (JPEG files can be converted to BMP by a variety of windows programs). The width and height of images should be a power of 2, for example 128 by 256, 16 by 64, 512 by 32, etc.

The 2d coordinates for a texture image range from 0.0 to 1.0. Coordinate are defined by an s axis and a t axis. The s-axis is horizontal and the t-axis is vertical with (s=0.0, t=0.0) located at the lower left corner of the image and (s=1.0, t=1.0) located at the upper right corner. (See Figure 50).
For texture coordinate values greater than 1.0 or less than 0.0 the mapping will be handled based on the values of the `texture_wrap_s` and the `texture_wrap_t` attributes. When an attribute is set to `_repeat` (which is the default), the pixel image is repeated as (s,t) values increase past 1.0 (or decrease past 0.0). If the texture wrapping attributes are set to `_clamp_to_edge`, the same pixel values found at [s,t] = 1.0 will be copied for all values of [s,t] > 1.0. Pixel values found at [s,t] = 0.0 will be repeated for [s,t] < 0.0.

Texture coordinates can only be specified when using objects derived from the `3dgraphic` class. The `3dgraphic`'`draw_texture_coordinate` method assigns a texture coordinate to the last vertex drawn with a `draw_vertex` method call. The `3dfaces` class allows texture coordinates to be specified in an array. These are described in the next chapter.

### 5.7.3 Front and back side visibility

In some cases it is necessary to specify which sides of a surface can be made visible. If the front or back side is always hidden, some performance improvement can be made by marking that side as such. The `visibility` attribute of `3dmaterial` controls which sides are visible (an invisible side will appear translucent when facing the 3dcamera). One of the constants `3dmaterial' _front`, `3dmaterial' _back`, or `3dmaterial' _front_and_back` can be assigned to the `visibility` attribute (`_front_and_back` is the default value).

```plaintext
let visibility(car_door_material) = 3dmaterial' _front
```
The _front_ side is defined by a counterclockwise winding of the vertices. The “right-hand thumb” rule can be used as mnemonic reminder. If a fist is made with the right hand and the vertices of a polygon are ordered in the direction the fingers point, then the thumb points out from the “front” of the surface.

### 5.7.4 Using instances of 3dmaterial

A caveat to using the 3dmaterial object is that each instance must be filed in a material_set before the object using it is rendered. 3dmaterials should be filed into the set owned by the 3dworld in which it will be visible. The 3dmodel also owns a material_set containing materials to be used within the model. When a 3dmodel is read in from a file, the material_set will be populated with all necessary 3dmaterial instances.

In the following example, a spinning cube is shown. The front and back surfaces reference a 3dmaterial showing a texture named “cacilogo.bmp”. Left and right side surfaces of the cube are displayed with the texture “eagle.bmp”. Top and bottom surfaces are colored with a light purple (diffuse_color = [1.0, 0.6, 1.0]). The cube itself is derived from the 3dbox class found in the 3dshapes.m subsystem. Its motion method is overridden to call the rotate_x, rotate_y, and rotate_z methods allowing the cube to spin about its x, y and z axes.

```plaintext
'Example 36: spinning cube with texture mapped surfaces. 
preamble including the 3d.m, 3dshapes.m subsystems
begin class spinning_box
  every spinning_box is a 3dbox and has
    a x_spin_rate, a y_spin_rate, a z_spin_rate and
    has a spin process method,
    overrides the motion

define x_spin_rate, y_spin_rate, z_spin_rate as double variable
end

define camera as a 3dcamera reference variable
define world as a 3dworld reference variable
end

method spinning_box'motion(dt)
  call rotate_x(x_spin_rate * dt)
  call rotate_y(y_spin_rate * dt)
  call rotate_z(z_spin_rate * dt)
end

process method spinning_box'spin
  let x_spin_rate = 90
  wait 4.0 units
  let x_spin_rate = 0
  let y_spin_rate = 180
  wait 10.0 units
  let y_spin_rate = 0
  let z_spin_rate = 360
  wait 4.0 units
end

main
```
define window as a 3dwindow reference variable
define light as a 3dlight reference variable
define box as a spinning_box reference variable
define lr_material, tb_material, fb_material as 3dmaterial reference variables

''create the window
create window
let title(window) = "Example 36: Showing a 2d image using texture mapping"

''create the world
create world
let ambient_color(world) = color'rgb(0.2, 0.2, 0.2)
file this world in world_set(window)

''create the camera
create camera
call set_perspective(camera)(60.0, 1.0, 0.1, 100.0, 1)
call set_location(camera)(0.0, 0.0, 2.0)
call set_orientation(camera)(0.0, 0.0, -1.0, 0.0, 1.0, 0.0)
file this camera in camera_set(world)

''create materials
create fb_material, lr_material, tb_material
let texture_name(fb_material) = "cacilogo.bmp"
let texture_name(lr_material) = "eagle.bmp"
let diffuse_color(tb_material) = color'rgb(1.0, 0.6, 1.0)
let ambient_color(lr_material) = color'rgb(1.0, 1.0, 0.0)
let visibility(fb_material) = 3dmaterial'_front
let visibility(lr_material) = 3dmaterial'_front
let visibility(tb_material) = 3dmaterial'_front
file this fb_material in material_set(world)
file this lr_material in material_set(world)
file this tb_material in material_set(world)

''create box
create box
let materials(box)(3dbox'_front) = fb_material
let materials(box)(3dbox'_back) = fb_material
let materials(box)(3dbox'_right) = lr_material
let materials(box)(3dbox'_left) = lr_material
let materials(box)(3dbox'_top) = tb_material
let materials(box)(3dbox'_bottom) = tb_material
let width(box) = 1.0
let height(box) = 1.0
let depth(box) = 1.0
file this box in motion_set
file this box in node_set(world)

''create the light
create light
let location_z(light) = 5.0
let location_y(light) = 2.5
file this light in light_set(world)

call display(window)

activate a spin(box) now
activate a animate(window)(30) now
let timescale.v = 100
start simulation
Figure 51: Example 36—A texture mapped cube
5.8 User Input

One of the advantages of a graphical simulation is the ability to interact with elements as the simulation runs. SIMSCRIPT 3d graphics provides allows the user to interact with the mouse, keyboard, graphics window frame and objects inside the window.

5.8.1 Mouse, keyboard and window frame interaction

In a SIMSCRIPT 3d program, the active or “top” 3d window can be thought to “receive” events from the mouse, keyboard, as well as the moving or resizing of the window frame. In order to receive mouse, keyboard or window events, the program must subclass the 3dwindow can override its action method. This method is called automatically in response to a user-driven event.

Another object found in 3d.m called 3devent is used by the action method. This object contains all data relevant event data. An instance is created by the runtime library and passed as the argument to the action method. The id attribute of 3devent is set to one of several predefined constants and describes the event that occurred. The list of possible events id constants is shown below:
<table>
<thead>
<tr>
<th>Id(event)</th>
<th>Cause</th>
<th>3devent attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>_activate</td>
<td>User clicked on window. Window brought to front.</td>
<td></td>
</tr>
<tr>
<td>_close</td>
<td>User clicked on the “X” to close the window.</td>
<td></td>
</tr>
<tr>
<td>_focus_in</td>
<td>Window now has the input focus.</td>
<td></td>
</tr>
<tr>
<td>_focus_out</td>
<td>Window has lost input focus.</td>
<td></td>
</tr>
<tr>
<td>_key_down</td>
<td>Pushing down a key on the keyboard</td>
<td>key_code</td>
</tr>
<tr>
<td>_key_up</td>
<td>Releasing a key on the keyboard</td>
<td>key_code</td>
</tr>
<tr>
<td>_mouse_down</td>
<td>Clicking in the canvas with the mouse.</td>
<td>x, y, modifiers, button_number, click_count</td>
</tr>
<tr>
<td>_mouse_up</td>
<td>Releasing the mouse button in the canvas.</td>
<td>x, y, modifiers, button_number, click_count</td>
</tr>
<tr>
<td>_mouse_move</td>
<td>Moving the mouse in the canvas.</td>
<td>x, y</td>
</tr>
<tr>
<td>_mouse_wheel_forward</td>
<td>Spin mouse wheel away from user (forward).</td>
<td></td>
</tr>
<tr>
<td>_mouse_wheel_backward</td>
<td>Spin mouse wheel toward user (backward).</td>
<td></td>
</tr>
<tr>
<td>_reposition</td>
<td>Dragging the window with the mouse.</td>
<td>x, y</td>
</tr>
<tr>
<td>_resize</td>
<td>Resizing the window with the mouse.</td>
<td>x, y</td>
</tr>
</tbody>
</table>
The action method should return one of the following two predefined constants: _continue or _block. If _continue is returned, the runtime library will handle the event. Returning with _block means that the runtime library will take no action in response to the event. For example, to keep the window from disappearing when closed by the user, the overridden action method should return with _block instead of _continue.

5.8.2 Mouse events

The action method can be used to receive mouse related events. The action method is called whenever the mouse is used within that window. Attributes of the 3devent instance passed to the action method are described in the tables below.

<table>
<thead>
<tr>
<th>Left, right or middle mouse button down click in canvas</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>3devent attribute</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>id</td>
<td>3devent’ mouse_down</td>
</tr>
<tr>
<td>x,y</td>
<td>Location in pixels of click from top left corner of canvas</td>
</tr>
<tr>
<td>button_number</td>
<td>1=left button, 2=middle button, 3=right button</td>
</tr>
<tr>
<td>click_count</td>
<td>1=single click, 2=double-click</td>
</tr>
<tr>
<td>modifiers</td>
<td>_shift_mod, _alt_mod, and/or _ctrl_moddepending on which key is held down during the click.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mouse movement in canvas</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mouse_up</td>
<td></td>
</tr>
<tr>
<td>_mouse_down</td>
<td></td>
</tr>
<tr>
<td>_mouse_move</td>
<td></td>
</tr>
<tr>
<td>3devent attribute</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------</td>
</tr>
<tr>
<td>id</td>
<td>3devent’ mouse move</td>
</tr>
<tr>
<td>x,y</td>
<td>Location in pixels of current pointer location from top left corner of canvas.</td>
</tr>
</tbody>
</table>

**Left, right or middle mouse button release in canvas**

<table>
<thead>
<tr>
<th>3devent attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>3devent’ mouse up</td>
</tr>
<tr>
<td>x,y</td>
<td>Location in pixels of mouse from top left corner of canvas</td>
</tr>
<tr>
<td>button_number</td>
<td>1=left button, 2=middle button, 3=right button</td>
</tr>
<tr>
<td>modifiers</td>
<td>_shift_mod, _alt_mod, and/or _ctrl_mod depending on which key is held down during the click.</td>
</tr>
</tbody>
</table>

**Mouse wheel rolled forward**

<table>
<thead>
<tr>
<th>3devent attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>3devent’ mouse_wheel_forward</td>
</tr>
<tr>
<td>x,y</td>
<td>Location in pixels of mouse from top left corner of canvas</td>
</tr>
<tr>
<td>button_number</td>
<td>Usually “2” to indicate the middle button</td>
</tr>
<tr>
<td>modifiers</td>
<td>_shift_mod, _alt_mod, and/or _ctrl_mod depending on which key is held down during the wheel movement.</td>
</tr>
</tbody>
</table>

**Mouse wheel rolled backward**

<table>
<thead>
<tr>
<th>3devent attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>3devent’ mouse_wheel_back</td>
</tr>
<tr>
<td>x,y</td>
<td>Location in pixels of mouse from top left corner of canvas</td>
</tr>
<tr>
<td>button_number</td>
<td>Usually “2” to indicate the middle button</td>
</tr>
<tr>
<td>modifiers</td>
<td>_shift_mod, _alt_mod, and/or _ctrl_mod depending on which key is held down during the wheel movement.</td>
</tr>
</tbody>
</table>

In the following example, we will modify the Example29 program to receive mouse movement. The program will show the model of a car and allow the user to change the orientation of the camera by clicking and dragging the mouse. Dragging the mouse left will rotate the camera counter-clockwise, and dragging right will rotate clockwise. Dragging up will rotate up and dragging down will rotate the camera down.

A “zoom” function is easily implemented by changing the perspective on the camera. Zooming in is accomplished by narrowing the depth of field view via the set_perspective method of the 3dcamera object. Zooming out can be done by widening depth of field. The mouse wheel is a natural tool for performing a zoom. Rolling the wheel forward should zoom in while rolling it backwards zooms out.
The program will define a sub-class of 3dwindow called *my_window* and override the *action* method. Inside *action*, the id attribute of the given 3d_event instance is compared with one of the following constants: *_mouse_up*, *_mouse_down*, *_mouse_move*, *_mouse_wheel_forward*, or *_mouse_wheel_back*. When the mouse is clicked down, the location is saved and a flag is set to indicate that “dragging” is on. When the mouse is moved and the “drag_on” flag is set, its current location (given in the 3d_event’s x and y attributes) is compared against the last location and camera is rotated accordingly. When the mouse is clicked up, we clear the “drag_on” flag. The code handling the *_mouse_wheel_forward* and *_mouse_wheel_back* events will call the 3d_camera’s depth of field via the *set_perspective* method to implement ‘zoom in’ and ‘zoom out’.

```
''example 37: Getting mouse input
preamble including the 3d.m subsystems
begin class my_window
  every my_window is a 3dwindow and has
    a drag_on,
    a mouse_x,  ''last mouse position
    a mouse_y,
    a phi,  ''camera angle about Y
    a omega,
    a dof_angle,
  overrides the action

define drag_on, mouse_x, mouse_y, dof_angle as integer variables
define phi, omega as double variables
end

define the_camera as a 3d_camera reference variable

define _nearp=1.0, _farp=10.0 as constants  ''near,far clipping planes
end

method my_window'action(event)
  ''check the event id against events to handle
  select case id(event)
    case 3devent'_mouse_down
      let drag_on = 1
      let mouse_x = x(event)
      let mouse_y = y(event)
    case 3devent'_mouse_move
      if drag_on <> 0
        add (x(event) - mouse_x) * 0.1 to phi  ''update camera angles
        add (mouse_y - y(event)) * 0.1 to omega
        call set_forward(the_camera)(0.0, 0.0, -1.0)  ''reset
        call rotate_y(the_camera)(phi)  ''rotate camera about its Y axis
        call rotate_x(the_camera)(omega)  ''now rotate camera about its X axis
        let mouse_x = x(event)  ''save last mouse position
        let mouse_y = y(event)
    always
    case 3devent'_mouse_up
      let drag_on = 0
    case 3devent'_mouse_wheel_forward
      ''zoom in by changing depth of field
      let dof_angle = max.f(dof_angle-2, 5)
      call set_perspective(the_camera)(dof_angle, 1.0, _nearp, _farp, 1)
    case 3devent'_mouse_wheel_back
      ''zoom out by changing depth of field
      let dof_angle = min.f(dof_angle+2, 175)
      call set_perspective(the_camera)(dof_angle, 1.0, _nearp, _farp, 1)
```

169
default
endselect

return with _continue ''tell SIMSCRIPT to do default handling
end

main
define window as a my_window reference variable
define world as a 3dworld reference variable
define light as a 3dlight reference variable
define model as a 3dmodel reference variable
define the_car as a 3dnode reference variable

''Create windows, worlds, and groups
create window
let title(window) = "Drag the mouse and use wheel to change the view"
let dof_angle(window) = 60

create world
file this world in world_set(window)

''create camera(s)
create the_camera
call set_forward(the_camera)(0.0, 0.0, -1.0)
call set_location(the_camera)(0.0, 0.0, 2.5)
call set_perspective(the_camera)(dof_angle(window), 1.0, _nearp, _farp, 1)
file the_camera in camera_set(world)

''create light(s)
create light
call set_location(light)(0.0, 500.0, 500.0)
file this light in light_set(world)

''create model(s) of some people
create model
let enabled(model)(3dmodel' smoothing) = 1
call read(model)("people.3ds", "")
file this model in model_set(world)

''create objects used in the simulation
create the_car
let model(the_car) = model
file this the_car in node_set(world)

''Call the display method of the 3dwindow.
''This will show the window.
call display(window)
activate a animate(window)(10000) now

let timescale.v = 100
start simulation
end

5.8.3 Keyboard input

Keyboard input can also be obtained by overriding the action method. The event id’s of interest are _key_up and _key_down. When these ids are given the key_code attribute of 3devent will contain the predefined constant representing the key. When key_code is set to the constant _literal_key, the key_literal integer attribute must be used to get the code.
The key_literal attribute will contain the character code of the key that was pressed. Essentially, the key_code attribute is used to handle function, arrow and other special keys while the key_literal attribute is used for the remaining alpha-numeric keys.

For example, suppose the previous example was to be modified to use the keyboard instead of the mouse to move the camera. In this case the action method would be changed to intercept keystrokes by comparing the 3devent’s id attribute with _key_down. The left, right, up, and down keys could be used to control the camera, so the key_code attribute should be compared with the _left_key, _right_key, _up_key and _down_key constants. We can also use the “i” and “o” keys to zoom in and out respectfully. When the key_code attribute is equal to _literal_key, the key_literal attribute will contain the alpha-numeric code. In our case, the attribute can be compared with text constants “o” and “i”. The action method implementation now looks like this:

```
method my_window'action(event)
    ''check the event id against events to handle
    select case id(event)
    case 3devent'-key_down
        select case key_code(event)
        case 3devent'-left_key   call move_camera(1.0, 0.0)
        case 3devent'-right_key  call move_camera(-1.0, 0.0)
        case 3devent'-up_key     call move_camera(0.0, -1.0)
        case 3devent'-down_key   call move_camera(0.0, 1.0)
        case 3devent'-literal_key
            select case key_literal(event)
            case "i"  call zoom_camera(-1.0)
            case "o"  call zoom_camera(1.0)
            default
                endselect
        endselect
    endselect
    default
        endselect
    return with _continue ''tell SIMSCRIPT to do default handling
end
```

5.8.4 Selecting a node in the scene-graph

It can be helpful to an application user to have the ability to interact with a graphical simulation. One of the nice things about adding mouse support to a simulation is the additional ability to click on visible objects in the 3dworld. SIMSCRIPT 3 graphics allows actions to be taken whenever any 3dnode in a scene-graph is clicked-on.

To support this, the 3dwindow is sub-classed and its action method overridden. The _mouse_down event is handled as shown above. The select_node method owned by the 3dworld class can then be called to determine which node was clicked on. Select_node takes the location in pixels of the mouse click and returns with the selected “leaf” node in the scene-graph. The location in pixels can be obtained from the x and y attributes of the 3devent instance (argument to action).
The `select_node` method will always return the leaf node in the scene graph that was selected. Many times the leaf node will compose low-level geometry that is not interesting to the application. The `get_owner_node` method of this leaf can be called to get the node that contains it in its `node_set`.

In the following, example10 will be modified to allow the user to click on the “people” model that is displayed. In the particular model, the people are represented by a scene-graph of body parts like “chest”, “head” and “abdomen”, etc. Our example will print the name of the body part that was clicked. Since the `select_node` method returns with a node in the visible scene-graph, it is important to note that the 3dmodel in this example must be “loaded” by calling the `3dnode'load` method instead of begin simply referenced (by assigning the `model` attribute). Otherwise there would only be one node in the visible scene-graph and that same node would be returned regardless of where the user clicks.

```
'example 38: Selecting a 3d object
preamble including the 3d.m subsystems
   begin class my_window
      every my_window is a 3dwindow and
      overrides the action
   end
   define the_message as a 3dtext reference variable
   define the_world as a 3dworld reference variable
end

method my_window'action(event)
   define node as a 3dnode reference variable
   ''look for a mouse down event
   if id(event) = 3devent'_mouse_down
      ''get which node in the scene-graph was selected. If it is an
      ''unnamed leaf, check its owner node for a name
      let node = select_node(the_world)(x(event), y(event))
      while node <> 0 and name(node) = ""
         let node = get_owner_node(node)
      while node <> 0
         let string(the_message) = "Selected node named: " + name(node)
         call update_drawing(the_message)(3dgraphic'_once)
      always
      always
      return with _continue ''tell SIMSCRIPT to do default handling for the
   end

main
   define window as a my_window reference variable
   define light as a 3dlight reference variable
   define model as a 3dmodel reference variable
   define the_people as a 3dnode reference variable
   define the_camera as a 3dcamera reference variable
   ''Create windows, worlds, and groups
   create window
   let title(window) = "Test of 3d component selection"
   create the_world
```

file this the_world in world_set(window)

'create camera(s)
call set_forward(the_camera)(0.0, 0.0, -1.0)
call set_location(the_camera)(0.0, 0.5, 1.5)
call set_perspective(the_camera)(60.0, 1.0, 1.0, 10.0, 1)
file the_camera in camera_set(the_world)

'create light(s)
call set_location(light)(0.0, 500.0, 500.0)
file this light in light_set(the_world)

'create model(s) of some people
call read(model)("people.3ds", "")
file this model in model_set(the_world)

'create objects used in the simulation
create the_people
call load(the_people)(model)
file this the_people in node_set(the_world)

'create a text message
call read_string(the_message) = "Click on the people!"
file this the_message in node_set(the_world)

'Call the display method of the 3dwindow.
' This will show the window.
call display(window)
activate a animate(window)(10000) now

let timescale.v = 100
start simulation
end
Figure 53: Clicking on the lady’s handbag
5.9 Animation and Simulation

Animating a 3d scene-graph comes naturally when running a simulation. Objects being simulated are usually moving or somehow changing shape over time. The application will define this dynamic behavior by implementing time-elapsing process methods that will update the location, rotation, geometry, etc of the visible scene-graph. The SIMSCRIPT III 3d graphics runtime library will automatically update the canvas of the 3dwindow that is showing the scene to concur with the attributes of the 3dnode objects in the scene-graph.

5.9.1 Frame Based Animation

This updating is done differently than in the gui.m 2d graphics. In a 2d graphics application, an attribute of a graphic object is assigned and the display method is then called to immediately update the appearance of that particular instance. The SIMSCRIPT 3d graphics instead uses a frame based system for updating the canvas. The image of each entire 3dworld scene-graph is updated as often as possible without causing a noticeable lag in the expected runtime of the simulation. The refresh rate depends on many factors including the complexity of the scene-graph, the time scaling factor, and the size and scope of the simulation.

To start this automated updating of the window, the 3dwindow animate process method must be activated after the 3dwindow is created. This process method takes the duration in time units as its only argument. If the duration is less than or equal to zero, animate will run forever. This process method must be running in order for mouse and keyboard events to work.

```
"Update the window automatically for 10000 units
activate a animate(window)(10000) now
start simulation
```

5.9.2 Time scaling

When a typical non-graphical SIMSCRIPT III simulation runs, TIME.V is automatically set to the time of the next pending process notice on the event set. If that process notices is scheduled to run 1000 units in the future, TIME.V will “jump ahead” to that time value. The behavior is not good for a graphical simulation because if time is not advancing smoothly, dynamic objects will “jump around” without regard to the speed at which they are supposed to move. For this reason, SIMSCRIPT provides a global variable called TIMESCALE.V. TIMESCALE.V is the number of 1/100ths of a second of real time per unit of simulation time. Setting TIMESCALE.V to 100 means that every unit of simulation time will elapse 1 second of real time. Therefore assigning a low value to TIMESCALE.V will speed up the motion of all moving objects. (Its use in 3d
graphics is identical the usage in 2d graphics). By default TIMESCALE.V is zero meaning that no time scaling is performed and TIME.V will advance without delay to the time of the next event. Every graphical simulation should assign this variable.

```plaintext
'I want to burn 2 real seconds for every 'UNIT' of simtime
let timescale.v = 200
```

### 5.9.3 Automatic motion and the motion_set

In a 3d graphics program attributes of a 3dnode object are assigned, but the update of its image is automatic as long as the 3dnode instance is filed into a *node_set*. For example, suppose we wanted to show a 3d object move in a straight line from (-2.0, 0.0, 0.0) to (2.0, 0.0, 0.0). A process method is written that, in a loop, calls the *set_location* method to update the position of the object, then waits a small amount of time called `_delta_x`.

```plaintext
process method car'move_it
    define x as a double variable
    define _delta_x=1.0, _speed=10.0 as constants
    for x = -20.0 to 20.0 by _delta_x
    do
        call set_location(x, 0.0, 0.0)
        wait _delta_x/_speed units
    loop
end
```

In the preceding example, the smoothness of the motion depends on the `_delta_x` constant. If this constant is made smaller, the motion will smooth out. However a problem with reducing this value is that it will increase the number of “wait” statement executions. For example, changing `_delta_x` from 1.0 to 0.01 will increase the number of iterations of the loop from 40 to 4000. If hundreds of objects were moving around, that could lead to millions of process switches just to support movement!

Fortunately, SIMSCRIPT III provides a better way to support animated motion. The 3d.m subsystem owns a set called *motion_set*. 3dnode instances can be filed into this set to have their movement updated automatically. The runtime library will call the *motion* method for every 3dnode filed into the *motion_set* before refreshing the window canvases. The default behavior of *motion* is to update the location of the 3dnode based on its velocity (which can be assigned by calling the *set_velocity*, or *move_to* methods). An application can override this method to provide customized motion such as rotation, or non-linear movement.

For example, suppose we want to use this technique instead of the process method to move the car from (-20,0,0) to (20,0,0). The *move_it* process method would be rewritten as below. Notice that using this technique only one “wait” statement is performed.

```plaintext
process method car'move_it
    define _speed=10.0 as constants
```
file this car in motion_set
call set_location(-20.0, 0.0, 0.0)
call set_velocity(_speed, 0.0, 0.0)
wait (20.0 - (-20.0)) / _speed units
remove this car from motion_set
end

For simple linear motion there is yet another way to move an object. The 3dnode class defines a process method called move_to. This process method will automatically compute the velocity and time to wait for the move. Using the move_to process method we no longer need the move_it method. The following code is added to the initialization:

file the_car in motion_set
call set_location(the_car)(-20.0, 0.0, 0.0)
activate a move_to(the_car)(20.0, 0.0, 0.0, 10.0) now

5.9.4 Moving Car Example

The following represents the simplest way to achieve linear motion that is driven by a simulation. A model of a car is initialized and its 3dnode move_to method is activated at time 0.0. This method will move the car from (-20.0, 0.0,0.0) to (20.0, 0.0,0.0) automatically during the simulation.

''Example 39:
''Example of moving a graphical object around in a simulation
''Requires ford.3ds model file
preamble including the 3d.m subsystems
end
main
define window as a 3dwindow reference variable
define world as a 3dworld reference variable
define camera as a 3dcamera reference variable
define light as a 3dlight reference variable
define model as a 3dmodel reference variable
define the_car as a car reference variable

create window
let title(window) = "Move a car across the screen"
create world
file this world in world_set(window)

create camera
call set_orientation(camera)(0.0, 0.0, -1.0, 0.0, 1.0, 0.0)
call set_location(camera)(0.0, 0.0, 25.0)
file this camera in camera_set(world)

create light
call set_location(light)(0.0, 500.0, 500.0)
file this light in light_set(world)

create model
call read(model)("ford.3ds", "")
file this model in model_set(world)
create the_car
let model(the_car) = model
call set_forward(the_car)(-1.0, 0.0, 0.0)
file this the_car in node_set(world)
call display(window)

'' activate the move_to method
file the_car in motion_set
call set_location(the_car)(-20.0, 0.0, 0.0)
activate a move_to(the_car)(20.0, 0.0, 0.0, 10.0) now ''destination, speed
activate a animate(window)(0) now

let timescale.v = 100 ''real seconds/ simulated seconds
start simulation
end

5.9.5 Achieving customized motion

Linear motion is not always adequate. The specific needs of the program to rotate, scale or move objects in a non-linear fashion over time may by complex and even depend on conditions in the simulation. To support this sub-classes of the 3dnode class can override the motion method. The motion method is called for all objects filed into the motion_set. Here the program can rotate, scale or set the location of the object based on the time differential “dt” passed as the argument to motion. Remember that the object must be filed into the motion_set and the simulation must be running before the motion method will be called. To stop the object from moving it should be removed from the motion_set.

The code below was taken from example13.sim and shows the implementation of the motion method. The "dt" argument is used in conjunction with application defined attributes such as airspeed and radius to determine the next location and orientation.

method moving_jet'motion(dt)
    add arcsin.f(min.f(pi.c / 2.0, airspeed * dt / flight_radius)) to flight_angle
    add vertical_airspeed * dt to elevation
    if elevation > _max_elevation or elevation < 0.0
        let vertical_airspeed = -vertical_airspeed
        always
    call set_location(flight_radius * cos.f(flight_angle), elevation,
        flight_radius * sin.f(flight_angle))
call set_forward(cos.f(flight_angle+pi.c/2.0), 0.0, sin.f(flight_angle + pi.c/2.0))
end

The complete example shown below will display 10 circling planes. Each plane moves in a slightly different manner at a different speed. The planes can be selected while the simulation is running. In this example, many of the concepts described in previous chapters are utilized including:
Overriding the 3dwindow’s action method to get mouse input, then calling select_node to determine which plane was clicked on.

- Overriding the 3dgraphic’s draw method to construct the planes with program code.
- Overriding the motion method to allow customized motion. Filing objects in the motion_set.
- Creating and utilizing a 3dtext object to display a message in the window.

''Example 40
''basic test of 3d graphics
''This program tests a simple simulation with many moving objects

preamble including the gui.m, 3d.m subsystems
begin class sky_window
  every sky_window is a 3dwindow and overrides the action
end

begin class moving_jet
  every moving_jet is a 3dgraphic and has
    a color,
    a flight_angle,
    a flight_radius,
    a airspeed,
    a vertical_airspeed,
    an elevation,
    a draw_triangle method,
    an init method and
  overrides the draw,
  overrides the motion,
  overrides the action

  define color as an integer variable
  define flight_angle, flight_radius, airspeed, vertical_airspeed,
  elevation
    as double variables
  define draw_triangle as a method given
    9 double arguments  ''vertices

    after creating a moving_jet call init
end

define the_window as a sky_window reference variable
define the_world as a 3dworld reference variable
define the_light as a 3dlight reference variable
define the_camera as a 3dcamera reference variable
define the_message as a 3dtext reference variable

define _max_elevation=2000.0 as a constant
end

''helper method to draw a single triangle
method moving_jet'draw_triangle(p1x, ply, plz, p2x, p2y, p2z, p3x, p3y, p3z)
  define nx, ny, nz as double variables

  call 3d.m:compute_normal_vector(p1x, ply, plz, p2x, p2y, p2z, p3x, p3y, p3z)
    yielding nx, ny, nz

  call draw_normal(nx, ny, nz)
call draw_vertex(p1x, p1y, p1z)
call draw_vertex(p2x, p2y, p2z)
call draw_vertex(p3x, p3y, p3z)
end

''The method is called by the system whenever it needs to know how to
''"draw" the jet.  code is provided to construct the geometry
method moving_jet'draw
  call set_color(_front, _ambient, color)
  call set_color(_front, _diffuse, color)
  call begin_drawing(_triangles)
  call draw_triangle(0.0, 0.0, 60.0, -15.0, 0.0, 30.0, 15.0,0.0,30.0)
  call draw_triangle(15.0,0.0, 30.0, 0.0, 15.0, 30.0, 0.0, 0.0, 60.0)
  call draw_triangle(-15.0,0.0, 30.0, 0.0, 15.0, 30.0, 0.0, 0.0, -56.0)
  call draw_triangle(0.0, 0.0, -56.0, 0.0, 15.0, 30.0, 15.0,0.0,30.0)
  call draw_triangle(15.0,0.0,30.0,-15.0,0.0,30.0,0.0,0.0,-56.0)
  call draw_triangle(0.0,2.0,27.0,-60.0,2.0,-8.0,60.0,2.0,-8.0)
  call draw_triangle(60.0, 2.0, -8.0, 0.0, 7.0, -8.0, 0.0,2.0,27.0)
  call draw_triangle(60.0, 2.0, -8.0, -60.0, 2.0, -8.0, 0.0,7.0,-8.0)
  call draw_triangle(0.0,2.0,27.0, 0.0, 7.0, -8.0, -60.0, 2.0, -8.0)
  call draw_triangle(-30.0, -0.50, -57.0, 30.0, -0.5, -57.0, 0.0,-0.50,-40.0)
  call draw_triangle(0.0, -0.5, -40.0, 30.0, -0.5, -57.0, 0.0, 4.0, -57.0)
  call draw_triangle(0.0, 4.0, -57.0, -30.0, -0.5, -57.0, 0.0, 0.0, -5.0, -40.0)
  call draw_triangle(30.0,-0.5, -57.0, -30.0, -0.5, -57.0, 0.0, 4.0, -57.0)
  call draw_triangle(0.0,0.5, -40.0, 3.0, 0.5, -57.0, 0.0, 25.0, -65.0)
  call draw_triangle(0.0, 25.0, -65.0, -3.0, 0.5, -57.0, 0.0,0.5,-40.0)
  call draw_triangle(3.0,0.5,-57.0,-3.0,0.5,-57.0,0.0,25.0,-65.0)
  call end_drawing
end

''initialize the flight parameters randomly
method moving_jet'init
  let flight_radius = uniform.f(100.0, 1500.0, 1)
  let flight_angle = uniform.f(0.0, pi.c * 2.0, 1)
  let airspeed = uniform.f(50.0, 500.0, 1)
  let elevation = uniform.f(0.0, 200.0, 1)
  let vertical_airspeed = uniform.f(-50.0, 50.0, 1)
end

''This method is called automatically by the system.  Using the "dt"
''argument we can determine the next location and orientation using the
''current elevation, airspeed, and vertical airspeed.
method moving_jet'motion(dt)
  add arcsin.f(min.f(1.0, airspeed * dt / flight_radius)) to flight_angle
  add vertical_airspeed * dt to elevation
  if elevation > _max_elevation or elevation < 0.0
    let vertical_airspeed = -vertical_airspeed
  always
  call set_location(flight_radius * cos.f(flight_angle), elevation,
                   flight_radius * sin.f(flight_angle))
  call set_forward(cos.f(flight_angle+pi.c/2.0),
                   sin.f(flight_angle+pi.c/2.0))
end

''We call this method from sky_window'action if a jet is clicked on.
method moving_jet'action(event)
  let event=event
  let string(the_message) = name + " was clicked on"
  call update_drawing(the_message)(3dgraphic'_once)
return with 0
end

'overriding the window's action methods lets us to get mouse clicks.
'If the user pushes the mouse button, call the "select_node" method
'to see if a plane was clicked on
method sky_window'action(event)
define node as a 3dnode reference variable
if id(event) = 3devent'_mouse_down
    let node = select_node(the_world)(x(event), y(event))
    if node <> 0
        call action(node)(event)
    else
        let string(the_message) = "Click on a plane..
        call update_drawing(the_message)(3dgraphic'_once)
    always
always
return with 0
end

main
define _num_planes=10 as a constant
define i as an integer variable
define plane as a moving_jet reference variable
define names as a 1-dim text array
reserve names(*) as _num_planes
let names(1) =  "Bobcat"
let names(2) =  "Prince"
let names(3) =  "Mercury"
let names(4) =  "Jumper"
let names(5) =  "Cowboy"
let names(6) =  "Flash"
let names(7) =  "Joker"
let names(8) =  "Iceman"
let names(9) =  "Reddog"
let names(10) = "Ace"

'create the window
create the_window
let title(the_window) = "Simulation and 3d graphics"
let color(the_window) = color'rgb(0.1, 0.2, 0.5)

'create the world
create the_world
file the_world in world_set(the_window)

'create the camera
create the_camera
call set_forward(the_camera)(0.0, 0.0, -1.0)
call set_location(the_camera)(0.0, 500.0, 1500.0)
call set_perspective(the_camera)(60.0, 1.0, 1.0, 3000.0, 1)
file this the_camera in camera_set(the_world)

'create the axis graphics
for i = 1 to _num_planes
do
    create plane
    let color(plane) = color'rgb(0.5, random.f(1), random.f(1))
    let name(plane) = names(i)
    file this plane in node_set(the_world)
file this plane in 3d.m:motion_set
loop

''create a text message using the 3dtext object
''give it a raster font so that its size will not depend on
''its distance from the camera
create the_message
call set_location(the_message)(-250.0, -250.0, 0.0)
let font(the_message) = 3dtext'times_roman_24_font
let string(the_message) = "Click on a plane.."
file the_message in node_set(the_world)

''simulate sunlight using a directional light from the top
create the_light
let ambient_color(the_light) = color'_black
let diffuse_color(the_light) = color'_white
let variety(the_light) = 3dlight'_directional
call set_forward(the_light)(0.0, -1.0, 0.0) ''point straight down
file this the_light in light_set(the_world)

let ambient_color(the_world) = color'rgb(0.0, 0.0, 0.3)
call display(the_window)

activate a animate(the_window)(1000) now
let timescale.v = 100 ''set the speed of the animation
start simulation
end
Figure 54: Example 40 output, planes circling in the sky

Flash was clicked on
6. Graphical Object Reference

The guide below contains an enumerated list of all objects used for graphical purposes. They are categorized by the subsystems for which they reside (gui, 3d, 3dshapes). Each heading indicates the inheritance relationship with the object base class(s) using the right arrow (→) to point to the base class.

6.1 gui.m Objects

clock (clock → graph → graphic → guiitem)
The clock class is used to display graphically the current simulation time. Like all objects derived from graph, its visual attributes are defined in SimStudio and it is loaded into a program by assigning its appearance attribute. Clocks can be either digital or analog. The set_time method is called to set the time displayed on the clock. A clock can be updated automatically if it is linked with a "display" variable through the use of a show statement. i.e.

''Preamble code
display variables include clocktime

''main routine
define my_clock as a clock reference variable
...
show clocktime with my_clock

(see Example 8 for the complete code)

Class members

set_time(hours, minutes, seconds)
Sets the time displayed on the clock given integer values for the hours, minutes and seconds.

color
A color object is never to be created but is instead a class containing methods relating to creating and using colors. A "color" in both 2d and 3d SIMSCRIPT is represented by a 32-bit integer value the encodes the RED, GREEN and BLUE components. Class methods of the color class can be used to provide or interpret that integer value. A color integer can be obtained by calling either the rgb or rgba methods given red, green and blue component values as percentages that range from 0 to 1.

let r = 0.5  ''set window background color to redish blue
let g = 0.0
let b = 1.0
let color(my_window) = color'rgb(r, g, b)
Using the red, green or blue methods you can get the value of a color component.

''print out the color of the title of a trace plot
let title_color = part_color(my_plot)(plot'_title_part, 0)
write color'red(title_color), color'green(title_color), color'blue(title_color)
as
"The color of the plot title is (", D(4,1), ",", D(4,1), ",", D(4,1), ")" /

The color class defines a few predefined colors for convenience. The can be used in place of the rgb method.

''set window background color to white
let color(my_window) = color'_white

Class members →

color'alpha [double]
(class method) Returns with the percentage of alpha (transparancy) in a color value (range: 0.0 to 1.0).

color'blue [double]
(class method) Returns with the percentage of blue in a color value (range: 0.0 to 1.0).

color'green [double]
(class method) Returns with the percentage of green in a color value (range: 0.0 to 1.0).

color'red [double]
(class method) Returns with the percentage of red in a color value (range: 0.0 to 1.0).

color'rgb(red, green, blue) [integer4]
Returns a color code value given the percentages of red, green and blue in the desired color. red, green and blue arguments range from 0 to 1. The following constants can be used in place of this method: _black, _white, _red, _green, _blue, _cyan, _magenta, _yellow, _grey.

color'rgba(red, green, blue, alpha) [integer4]
Returns a color code value given the percentages of red, green, blue and and "alpha" code used for transparency (future use only) in the desired color. Arguments range from 0 to 1.

dialogbox (dialogbox → form → guitem)
The DialogBox object is derived from the Form object and represents both modal and modeless dialogs. Dialog boxes provide an interactive way for the user to enter input data. A dialog box is a window containing a variety of input controls including buttons,
text labels, tabular controls, single and multi-line text, combo, value, list, radio, and check boxes. Tabbed dialogs can also be created in SIMSCRIPT.

The layout of the fields in a DialogBox is done in the SimStudio dialog box editor. In the application, a dialog is initialized by assigning the appearance attribute to a template located using the name the dialog was given in SimStudio.

Every dialog box owns a set of field objects – each of which represents an item shown in the dialog (i.e. check box, value box, button, etc.). These field objects have methods that allow the program to set and get the data values shown in the dialog. See the reference for the field object.

Class members →

**accept_input [field reference]**

Call this method to allow the user to interact with the dialog box. For modal dialog boxes, this method will return when the user presses a 'terminating' button. A reference pointer to the selected field is returned. For modeless dialogs, the "display" method should be used show the dialog which will promptly return. If marked as "Asynchronous" in SimStudio, the current process is suspended.

**location_x [double]**

Set/get the x location of dialog box with respect to the screen. Dialogs are positioned like windows with (0,0) corresponding to the lower left and (32767,32767) corresponding to the upper right corners of the computer screen. The "dialog properties" box in SimStudio is used to control which corner of the dialog is positioned.

**location_y [double]**

Set/get the y location of dialog box with respect to the screen. Dialogs are positioned like windows with (0,0) corresponding to the lower left and (32767,32767) corresponding to the upper right corners of the computer screen. The "dialog properties" box in SimStudio is used to control which corner of the dialog is positioned.

**field (field → guiitem)**

The Field object provides an interface for passing data back and forth between the executing program and an item (check box, text box, etc) in a form. Field objects must be filed into the field_set owned by Form. In most cases, the Field object is created and filed into this set automatically at the time the appearance attribute is assigned. A corresponding field is created for each item in a dialog box, each menu item in a menu bar, and each button in a palette.

The name attribute of the field is initialized to the “Field name” given to the corresponding item in the SimStudio editor. This allows the program to obtain a reference to a particular field given its name. The find method defined by the Form class does just that. Given the text “name” of a field, Form find will return the Field reference value for the item.
The Field object defines several methods that can be used on the left or right side of an assignment. When the method is used on the left, the text, value or selection state shown by the corresponding item in the dialog box will change to show the new value. When the method is used on the right, the current text or value typed in by the user will be returned by the method.

The accept_input method of the DialogBox object will return with the Field reference pointer for the terminating button that was selected to dismiss the dialog. The name attribute of this Field can be inspected to determine which button was clicked on. Note that if the user dismisses the dialog box by clicking on the “X” in the window frame, accept_input will return with “0”. Therefore the application should check the return value before attempting to inspect the name of the returned field.

Class members →

activated [integer] {lr}
When used on the left sets the activation status of the field. Assign to '0' to grey out the field. (DEFAULT: 1)

field_set [set of field reference objects]
Fields created as the result setting the "appearance" attribute are "automatically filed into this set. Fields created dynamically must be filed in the application code.

find(field_text_id) [field reference] {r}
Recursively searches the field_set for the field given its text ID (assigned in SimStudio). The reference to this field is returned, or zero is returned if the field cannot be located.

image [text] {l}
Name of the file containing the bitmap image for the item. (Currently only tree items have a bitmapped image (DEFAULT: "")

label [text] {lr}
The label of the field can be retrieved or set. Should be called after "appearance" of parent form is assigned.

name [text] {lr}
Each field in a form has a unique name. This name will match the name given to the corresponding control in the dialog box editor. (DEFAULT: assigned in SimStudio editor.

scroll(row_number, column_number) {r}
Scrolls the list to include the specified item. This method only applies to fields that include a list of some kind.

selected [integer] {lr}
Gets/sets the current selection state of an item in the field. '1' means selected, while '0' means unselected. (DEFAULT: 0)

\textit{\textbf{selected\_at(row, column) [integer] {lr}}} \\
Gets/sets the current selection state of a list item in the field. The field should be a table, list box, or radio box, and the first argument refers to the row number of the item or button. For tables, row '0' refers to the column headers. The second argument is ignored unless the field being referred to is a 2-dim table. 1 => selected, 0 => unselected. (DEFAULT: 0)

\textit{\textbf{string [text] {lr}}} \\
Gets/sets text data in the field. Returns the text displayed in a text or combo box. If used on the left, sets the text. (DEFAULT: "")

\textit{\textbf{string\_at(row, column) [text] {lr}}} \\
Gets/sets text item in a field given the row and column of the item. Returns the text displayed in a text or combo box. If used on the left, sets the text. (DEFAULT: "")

\textit{\textbf{strings [pointer] {lr}}} \\
Gets/sets an array of text data in the field. If used on the right, returns the array of text contained in a list. If used on the left, the items will be set. A 1-dim text array should be used. (DEFAULT: 0 or assigned in SimStudio.)

\textit{\textbf{value [double] {lr}}} \\
Gets/sets numerical data in the field. If used on the right, returns '1.0' if a check box, is checked. Returns the value shown is a progress bar or value box. Returns the numeric value of the text in a text box. (DEFAULT: 0.0)
filebox (filebox → form → guiitem)
A filebox is a simple "system" dialog box that does not contain fields and is not defined in SimStudio. Its purpose is to show a file selection dialog.

Class members →

\begin{itemize}
\item \textbf{accept_input [integer]} \{r\}
  
  Call this method to display the dialog and let the user pick a file. Returns '1' if the user has made a selection, '0' if cancelled.

\item \textbf{file [text]} \{lr\}
  
  Retrieves the file name (without path) that was selected by the user.

\item \textbf{filter [text]} \{l\}
  
  set the wild card filter that will be shown with the dialog. (e.g. '*.dat') assign this property before calling accept_input if a filter is needed.

\item \textbf{path [text]} \{lr\}
  
  Can be assigned before calling accept_input to set initial path displayed in the dialog. Can be used on the right to get the path to the selected file after the call to accept_input.
\end{itemize}

fillstyle
Simple attribute container object that is used to hold fill style information for circles and polygons.

Class members →

\begin{itemize}
\item \textbf{pattern [integer]} \{lr\}
  
  Can be assigned to one of the following constants: _hollow, _solid, _narrow_diagonal, _medium_diagonal, _wide_diagonal, _narrow_crosshatch, _medium_crosshatch, _wide_crosshatch. (DEFAULT: _hollow).
\end{itemize}

form (form → guiitem)
The form class is the base class for the dialogbox, menubar, palette. Before being displayed, objects derived from form should be filed into the "form_set" owned by the window that is to contain it. (A form should never be created directly, only create instances of classes derived from form.) All forms has the following features:

1) Can receive input from the user. Immediate notification is possible.
2) Appearance is obtained from a Template created by SimStudio.
3) Owns a set of field objects, each corresponding to a control in the dialog. These objects are usually created at the time a form is loaded from a .sg2 file by assignment of the appearance attribute.

**Class members →**

`action(event) [integer] {n}`
This method must be overridden to obtain immediate notification of a change made to a field in the dialog box. Event data for this field is passed as the "event" argument. After handling the notification this method should return one of the following values:

- `_continue`
  Accept the input and continue

- `_terminate`
  Terminate the interaction and return from `accept_input`

`appearance [template reference] {l}`
Use on the left to specify the template that refers to a dialog, menu bar or palette created in the SimStudio editor. When this attribute is assigned, fields will be created for control items contained in the template and automatically filed into the `field_set`.

`field_set [set of field reference objects]`
Set of field reference objects owned by the form, each of which represents an item in the form, (such as a check box, radio button, etc.).

`find(field_id_text) [field reference] {r}`
Searches for the field given its text ID (assigned in SimStudio). The search is recursive, in other words the `field_set` owned by sub-fields will also be searched.

`title [text] {lr}`
Use on the left to specify the title shown in the title bar of a dialog or palette window.

**formevent (formevent → guievent)**
A formevent reference is passed to the action method by the SIMSCRIPT III runtime library at such time the user initiates the event. It contains information about the user's action on the field.

**Class members →**

`drop_x [double] {r}`
Location inside the canvas of the release point (in world coordinates of view upon which the object was dropped).

`drop_y [double] {r}`
Location inside the canvas of the release point (in world coordinates of view upon which the object was dropped).

*drop_view [view reference] {r}*

Top most view upon which the object was dropped.

*field [field reference] {r}*

Reference to the field object that was clicked on.

*guievent'id [integer] {r}*

The inherited guievent'id attribute will contain one of the following values

- **_button_dropped**
  - A dragable palette button was dropped
- **_button_pushed**
  - A button or menu item was pushed
- **_close**
  - User clicked to close the dialog
- **_data_changed**
  - Text or numerical data updated

**graph (graph → graphic → guiitem)**

A graph is the base class for all presentation charts and meter objects such as the 2d chart, piechart, dial, clock etc. Every graph should be filed into the "graphic_set" of a view object before being displayed. The graphs are designed in SimStudio, saved to the ".sg2" file, then loaded into the graph object by assigning the appearance attribute.

**Class members →**

*appearance [template reference] {l}*

Use on the left to specify the template that refers to a graph saved by the SimStudio Graph Editor.

*title [text] {lr}*

Use on left/right side to set/get title shown on graph. Should be assigned AFTER the appearance attribute.

**graphic (graphic → guiitem)**

The graphic object provides the base class for all objects that can both appear in the canvas, and can be composed of a group of shape primitives. This includes the icon and graph objects. A graphic can be created and used directly if customized drawing is required (i.e. the object's image is not known until runtime – the object cannot be edited using the SimStudio icon editor.) A graphic object (and its derived classes) can be made to appear inside a particular window by filing it into the graphic_set owned by one of the view objects belonging to the window.
For use as a stand-alone object, the `graphic` has several methods for drawing various shapes (these method names begin with `draw_`). Calls the these "draw" methods should be made between calls to the `begin_drawing` and `end_drawing` methods.

**Class members →**

`action(graphicevent) [integer]`
Called by SIMSCRIPT runtime whenever the user initiates some kind of action regarding the object. This methods should not be called directly, but should be overriden to obtain notification of the user interaction. This method should be implemented to return one of the following constants:

- `_continue`
  Process the click normally generating a WindowEvent.

- `_block`
  Do not generate a WindowEvent

`begin_drawing`
Calling this method will mark the beginning of a segment of the `graphic`'s image. The image will be defined using a sequence of calls to `draw_` methods. `begin_drawing` must be called before any of the "draw_" methods can be called. The `end_drawing` method should be called to mark the end of this segment of the image.

`display_at(x,y)`
This is shortcut method that will set the location_x and location_y attributes using the 2 given double arguments, then call the `display` method.

`draw_arc(x_center, y_center, radius, start_angle, end_angle, color, linestyle)`
Draws a semi-circular arc given its center point, radius, start and stop angles (in radians), color and line style. The arc is drawn counter-clockwise starting at the "start_angle" and stopping and "end_angle". The color of the arc is provided and it can be a solid or "dashed" line depending on the "linestyle" argument. (See `linestyle`).

`draw_circle(x_center, y_center, radius, color, fillstyle)`
Adds a circle to the drawing. The `x_center`, `y_center`, and `radius` arguments define the geometry. The `fillstyle` argument specifies if the circle will be hollow, solid, or have a pattern.

`draw_pie(x_center, y_center, radius, start_angle, stop_angle, color, fillstyle)`
Draws a filled pie slice given its center point, radius, and start and stop angles (in radians). The pie sector is drawn counter-clockwise starting at the "start_angle" and stopping and "end_angle". The `fillstyle` argument specifies if the slice will be hollow, solid, or have a pattern.
**draw_polygon(2d_points_array, color, fillstyle)**

Adds an n-sided polygon to the drawing. Points are given as a (2 x n) 2-dim double array. (points(1,*) references the "x" values, points(2,*) references "y" values.) The "color" is an integer obtained from the color'rgb method. The fillstyle argument specifies if the polygon will be hollow, solid, or have a pattern. The last point is automatically connected to the first.

**draw_polyline(2d_points_array, color, linestyle)**

Adds an n-sided polyline to the drawing. Points are given as a (2 x n) 2-dim double array. (points(1,*) references the "x" values, points(2,*) references "y" values.) The "color" is an integer obtained from the color'rgb method. The linestyle argument specifies if the lines will be solid, or dashed. First and last points are not connected.

**draw_polymark(2d_points_array, color, markstyle)**

Adds n single point markers the drawing. Points are given as a (2 x n) 2-dim double array. (points(1,*) references the "x" values, points(2,*) references "y" values.) The "color" is an integer obtained from the color'rgb method. The markstyle argument specifies which of several predefined shapes will be used for the marker.

**draw_rectangle(x_ll, y_ll, width, height, color, fillstyle)**

Adds a rectangle to the drawing. The first two arguments specify the (x,y) location of the lower left corner. Width and height are also given. The "color" is an integer obtained from the color'rgb method. The fillstyle argument specifies if the polygon will be hollow, solid, or have a pattern.

**draw_text(x_center, y_center, color, textfont, text_string)**

Draws a graphic text string. The font, point size, and alignment attributes are provided in the textfont reference argument. The center point of the text is given by (x_center, y_center). The "color" is an integer obtained from the color'rgb method. The text_string argument is the text to draw.

**get_bounding_box yielding (xlo, xhi, ylo, yhi)**

Computes the smallest rectangle necessary to completely enclose the graphic drawing. The bounding box is given in NDC coordinates.

**location_x [double] {lr}**

Can be used to get or set the x location in world coordinate units. (DEFAULT: 0.0)

**location_y [double] {lr}**

Can be used to get or set the y location in world coordinate units. (DEFAULT: 0.0)

**rotation [double] {l}**
Sets the current rotation in radians. Positive values specify a counter-clockwise rotation. DEFAULT: 0.0

**scale [double] {l}**
Sets the scaling factor. (A scaling factor of 2.0 would double the size of the object) (DEFAULT: 1.0)

**graphicevent (graphicevent → guievent)**
A reference to a *graphicevent* is passed to the *graphic'action* method after user clicks on the graphic or otherwise affects a graphic object while the simulation is running. The object instance is managed by the SIMSCRIPT runtime library and does not need to be created or disposed of by the application.

**Class members →**

**button_number [integer] {r}**
Ordinal position of the mouse button that was pressed to cause the event.

**click_count [integer] {r}**
"1" if single click, "2" if double click or "3" if triple click.

**guievent:id [integer] {r}**
Holds a constant that indicates the type of event that happened.

  _mouse_down
  A mouse button was pushed while the pointer was over the graphic.

  _mouse_up
  Mouse button let up while pointer is over the graphic.

  _mouse_enter
  Mouse pointer has just been moved over the top of the graphic.

  _mouse_leave
  Mouse pointer has just been moved away from the graphic.

**modifiers [integer] {r}**
ORed product of _shift_down, _ctrl_down and _alt_down. To test, use one of the constants as a mask parameter to the *and.f* function>

    if and.f(modifiers(graphicevent), _shift_down)
         write as "the shift key is down!", /
    endif

  _shift_down
  One of the <shift> keys was held down when the mouse was clicked.

  _ctrl_down
  One of the <control> keys was held down when the mouse was clicked.

  _alt_down
One of the <alt> keys was held down when the mouse was clicked.

**guievent**
A guievent is the base class for all events. An event is used to immediately inform the program of an action performed by the user. guievent object references are created by the SIMSCRIPT runtime library and not be created by the application.

**Class members →**

* id [integer]*
This code is to indicate the type of event. Codes are defined in the various sub-classes of guievent.

**guiitem**
The guiitem class is the base class for all gui.m classes that represent a visible graphic object such as icons, graphs and forms. (Derived objects can all be displayed and erased.) A guiitem should not be created directly.

**Class members →**

* display*
Displays the graphic.

* erase*
Erases the graphic.

* hidden [integer] {l}*
Assign this attribute to "1" to hide a graphic. Graphic must be un-hidden by assigning the attribute to "0" before it can be redisplayed.

* visible [integer] {r}*
Indicates "1" if guiitem is currently visible.

**icon (icon → graphic → guiitem)**
The icon object is derived from the graphic object and is used to graphically represent any moving or static object inside a window. Like the graphic objects, the icon is composed of a group of shapes such as lines, polygons, text. In addition, an icon object has an appearance property that enables its shapes to be loaded in from the graphics file created by SimStudio. The SimStudio Icon Editor’s JPEG image import feature allows an icon to be shown by a raster image. This allows photographs and images created by other programs to be shown in a SIMSCRIPT III application. Another important feature of the icon is its ability to have motion over simulation time. A velocity can be defined
for the icon, which will cause its position to be updated automatically as simulation time is advanced. Before being displayed, the icon should be filed into a "graphic_set" owned by a view object.

Class members →

appearance [template reference] {l}
Use on the left to specify the template that refers to an icon saved by the SimStudio Icon Editor. See the template class reference for instructions on how to load a template saved by the SimStudio editor.

color [integer] {l}
By assigning this monitored attribute various shapes within the icon can be recolored at runtime. The color will be applied only to shapes that are marked as "Color definable using DCOLOR.A". This is done by selecting the shape in the icon editor, clicking "Edit/Object Properties...", then checking the box in the dialog.

label [text] {lr}
Left usage: sets the 'definable' text in the icon. Right usage: gets the 'definable' text in the icon. A textual element of an icon can be changed at runtime by assigning the label property. Before this will work, the text element must be marked as "definable" in the icon editor. This is done by selecting the text, clicking "Edit/Object Properties...", then checking the box in the dialog called "Text definable using DTVAL.A".

motion(double)
Override this method to provide a custom motion to the icon. If the icon has a non-zero x or y velocity, this method will be called to compute the next location. By default, the location is computed assuming simple linear motion.

set_speed_course(speed, radians)
Sets the speed and course with which the icon is moving. velocity_x, and velocity_y, properties will be computed. Implicitly invokes start_motion.

start_moving
Can be used to add the icon to the set of moving objects. When in motion, the motion method is called repeatedly as simulation time elapses, and the position of the icon in the window is updated based on the velocity_x and velocity_y attributes. If the icon is already moving, nothing happens.

stop_moving
Can be used to remove the icon from the set of moving objects. When in motion, The "motion" method will no longer be called.

velocity_x [double]
Left usage: sets the current x velocity component. The object must be filed into a
`motion_set` by calling `start_moving` in order for velocity to take effect. (DEFAULT: 0.0)

`velocity_y [double]`
Left usage: sets the current y velocity component. The object must be filed into a
`motion_set` by calling `start_moving` in order for velocity to take effect. (DEFAULT: 0.0)

**linefont (linefont -> textfont)**
The `linefont` class is used to represent a vectored text font. Vector fonts can be arbitrarily
scaled and rotated in any direction. When attached to an `icon`, text graphics using this
font will scale and rotate with the icon.

**Class members →**

`angle [integer] {rl}`
Defines rotation of text in 0 to 360 degrees. DEFAULT: 0

`font [integer] / {rl}`
Predefined vector font name. Use one of the `linefont` class constants shown
below. (DEFAULT: `_basic`)

- **_basic**
  Rudimentary font. All characters have the same width.

- **_simple**
  Easy to read font. Variable width characters

- **_roman**
  Vector font that is similar to Times Roman. Variable width characters.

- **_bold_roman**
  Bold version of the `_roman` font.

- **_italic**
  Italicized version of the `_roman` font.

- **_script**
  Cursive font. Variable width characters.

- **_greek**
  Font that uses Greek characters in place of roman.

- **_gothic**
  Variable width gothic looking characters.

`size [integer] / {rl}`
Height of text characters in NDC coordinate units. (32768 is the maximum =
window canvas height). (DEFAULT: 560)
linestyle
Instances of this class are used to represent a set of specific line drawing attributes. Note that at initialization time the linestyle global variable is assigned a default linestyle instance.

Class members —

*pattern [integer] {rl}*
Indicated the dash pattern of the line. (DEFAULT: _solid)

  *solid line pattern:*
  _solid: A solid, unbroken line.

  *dashed line patterns:*
  The following patterns are self-explanatory: _long_dash, _dotted, _dash_dotted,
  _medium_dash, _dash_dot_dotted, _short_dash, _alternate.

*width [integer] {rl}*
Width of the line in NDC units (where 32768 is the maximum value). "0" indicates a 1-pixel wide line. (DEFAULT: 0)

markstyle
Instances of this class are used to represent a set of specific poly-marker drawing attributes. Note that at initialization time the markstyle global variable is assigned a default markstyle instance.

*size [integer] {lr}*
Size in NDC units (where 32768 is the maximum value). (DEFAULT: 500)

*type [integer] {lr}*
Identifies the shape of marker. Use one of the constants. (DEFAULT: _dot)

  Marker type constants
  One of these constants can be assigned to the type attribute of the markstyle class. _dot, _cross, _asterisk, _square, _x, _diamond

messagebox
A messagebox is a simple dialog box that does not contain fields. Its purpose is to provide simple information to the user or to ask a question. It can be loaded from a template by setting the appearance property. It can also be used without setting the appearance. Messages can be single or multi-line.

*accept_input [integer]*
Call this method to allow the user to interact with the message box. Returns with one of the following constants:

  Button constants (from accept_input)
One of the following constants will be returned depending on which button the user click: _ok_button, _cancel_button_yes_button, _no_button, _abort_button, _retry_button, _ignore_button.

message_line [text] {l}
This attribute can be assigned to the message text when only a single line of text is to be shown.

message_lines [pointer] {l}
Assign this attribute to a 1-dim text array pointer containing the text of the message to display.

meter (meter → graph → graphic → guiitem)
The meter class is used to show the value of a single display variable. After initializing the meter object instance, whenever the display variable that it is linked to is assigned, the meter will be automatically updated.

Class members →

set_value(value) {r}
Sets the meter to show the value of the given double argument.

palette (palette → form → guiitem)
A Palette is attached to the left, right, top, or bottom of a window frame. Each palette contains row(s) of square buttons each of which displays a small bitmap image. Palettes are used asynchronously – i.e. the program will receive notification of a button click via the overridden form'action method. The palette object is derived from form so therefore an instance of a palette object must be filed into the form_set owned by its window.

piechart (piechart → graph → graphic → guiitem)
The piechart represents a pie shaped graph. It is derived from the graph class and inherits graph properties (for example, it is constructed in the SimStudio graph editor and initialized by assigning its appearance attribute to a template loaded from the .sg2 file.

Class members →

set_slice(slice_number, slice_value)
Call this method to set one of the values shown in the chart. The given slice_number should range from 1 to the number of slices in the chart. The percentage of the pie occupied by the slice is the given slice_value divided by the sum of all slice values.
plot (plot → graph → graphic → guiitem)

The plot class is used to display a 1-dim array, 0-dim histogram or 0-dim value over simulation time. As with other objects derived from graph, it is constructed in the SimStudio graph editor and loaded from the .sg2 file when the appearance attribute is assigned. Note that "Properties" are reset when "appearance" is assigned.

Class members →

   add_data(data_set_number, x, y)
   Add a new data point to the plot. The chart will reflect the change when the guiitem'display" method is called. The "data_set_number" (integer) argument should range from 1 to the number of datasets.

   clear_data(data_set_number)
   Clears out all data points in the plot given the data set number to clear.

   ds_count [integer] {lr}
   Number of datasets in the chart. ds_count should be assigned before assigning any other dataset related properties or plotting any data. (DEFAULT: set in SimStudio).

   ds_interval(data_set_number) [double] {lr}
   Sets the width of the bars in the given dataset along the X axis of the plot. (DEFAULT: same as interval_x, the tic mark interval)

   ds_option(data_set_number, option_code) [integer] {lr}
   Sets the one of the data set options to the given data set option constants (show below).

      _ds_discrete
      When non-zero, there will be a fixed number of equally sized, successive data cells in the data set. If zero, the plot does not contain data "cells" – instead when plotting is done, a new point is shown in the plot.

      _ds_static
      Setting the _ds_static option to 'true" may result in less memory usage for cases where the values shown in the plot are plotted at a single instance, and after that no more plotting is done.

      _ds_surface
      A "surface" chart is shown instead of a bar graph – Plotted points (or data cells) along the x axis are directly connected in succession to form a jagged surface.

      _ds_centered
      For discrete plots only. When non-zero, vertical bars in the chart are laid out along the x-axis with respect to the center point of their bottom edge. For example, a bar located at x = x_min will actual cover from (x_min – x_interval / 2) to (x_min + x_interval / 2).
_ds_narrow
For discrete plots only. When non-zero, vertical bars are slightly thinner than
the data set interval width (ds_interval).

_ds_usey2axis
When non-zero data points will be scaled with respect to the second Y axis (if
present)

_ds_interpolate
Use linear interpolation to compute the y values at the discrete interval points
along the x-axis.

_ds_fill
Bars will be shown in the plot

_ds_line
Line segments will connect each data point

_ds_mark
Each data point will be shown with a marker.

ds_title(data_set_number) [text] {lr}
Sets the title of a data set (displayed in legend).

grid_interval_x, grid_interval_y, grid_interval_y2 [double] {lr}
Grid line interval for vertical grid lines populated along x, y and y2 axes.

min_x, min_y, min_y2 [double] {lr}
Minimum values along x, y or second y axes.

max_x, max_y, max_y2 [double] {lr}
Maximum value along x, y, or second y axes.

minor_interval_x, minor_interval_y, minor_interval_y2 [double] {lr}
x and y tic mark intervals for the smaller tics.

interval_x, interval_y, interval_y2 [double] {lr}
x and y major tic mark interval widths. x interval is also the cell width.

intercept_x, intercept_y, intercept_x2 [double] {lr}
intercept_x: point on x axis where y axis crosses
intercept_y: point on y axis where x axis crosses
intercept_x2: point on x axis where y2 axis crosses

num_interval_x, num_interval_y, num_interval_y2 [double] {lr}
x and y numbering intervals.

option(option_constant) [integer] {lr}
Chart options apply to the whole chart, not just an individual data set or data
point. When used on the left, the option should be assigned either "1" to enable or
"0" to disable the feature described by the "option_constant" argument. The following options are currently available:

_adjacent
Applies only to plots with multiple data sets. If enabled, bars for successive data sets are not stacked but made thinner and shown side by side in the same interval.

_compress
Used for time trace plots only. Controls how the plot behaves when simulation time exceeds the maximum X value. If enabled, ONLY the max_x boundary of the plot is increased when time exceeds the current max_x attribute value. All previously plotted data is retained. If zero, a moving window is used. In other words both the min_x and max_x boundaries are increased to encompass the new sim. time. Old data is discarded.

_rescale_x, _rescale_y, _rescale_y2
If enabled, plotting values outside the boundaries of the plot will force the plot to resize (i.e. adjust x_max, y_max, etc.). The size will change enough to allow the new point to appear in the plot.

_show_box
If enabled, a box is drawn around the outside of the plot

_show_legend
If enable, a legend will be shown at the bottom of the plot. Data set title (ds_title) text is displayed.

_show_x_grid, _show_y_grid, _show_y2_grid
If enabled, grid lines are shown.

_time_trace
Should be enabled if the plot is a time trace plot linked to a single display variable. When the display variable changed value, the new value is plotted using the current simulation time as the "x" axis value.

_x_tics_inside, _y_tics_inside
Tic marks are drawn from the axis toward the interior of the plot.

_x_tics_outside, _y_tics_outside
Tic marks are drawn from the axis toward the exterior of the plot.

<parts>

part_color(part_constant, data_set_number) [integer] {l}
Sets the color of the given part of the plot. The "data_set_number" may or may not be required.
part_fillstyle(part_constant, data_set_number) [fillstyle reference] {l}
Sets the fill style of the given part of the plot. The "data_set_number" may or may not be required.

part_linestyle(part_constant, data_set_number) [linestyle reference] {l}
Sets the dash (line) style of the given part of the plot. The "data_set_number" may or may not be required.

part_markstyle(part_constant, data_set_number) [markstyle reference] {l}
Sets the marker style used to represent data points. The "data_set_number" is required.

part_font(part_constant, data_set_number) [textfont reference] {l}
Sets the font info for text elements.

part_hidden(part_constant, data_set_number) [integer] {l}
Assign "1" to hide a part.

plot_data(data_set_number, x, y)
Add a new data point to the plot. The chart will reflect the change immediately. The "data_set_number" (integer) argument should range from 1 to the number of datasets.

title_x, title_y, title_y2 [text] {lr}
Set the titles shown on x and y axes.

popupmenu (popupmenu → form → guiitem)
Popup menus allow an end user to right click in the canvas of a window to display a small menu showing a list of choices. They are sometimes referred to as context menus because a different menu can easily be displayed depending on both where and when the user right-clicks in the window. Must be filed into the form_set owned by the window where it lives.

Class members →

accept_input [field reference] {r}
Displays the popup menu and waits for the user to make a selection. A field reference object pointer representing the selected menu item is returned. Zero is returned if the user hit the escape key rather than making a selection.

systemfont (systemfont → textfont)
This is a type of text font that refers to raster fonts. This font object represents the standard fonts found with the operating system. The size of a text primitive that is assigned one of these fonts is determined by the point_size attribute, and it will not scale in size with its icon.
**bold [integer] {lr}**
1 to indicate a bold face. 0 indicates plain text. (DEFAULT: 0)

**direction [integer] {lr}**
One of four angles for the text (DEFAULT: _right)

**Values for font direction**
The following constants can be assigned to the direction attribute: _right, _up, _left, _down. _left would mean normal "left-to-right" text strings. For _right, the text string would be rotated 180 degrees (not horizontally mirrored).

**family [text] {lr}**
The family name corresponds to the name of one of many fonts found on either MS Windows or Linux systems (i.e. "Arial", "Courier", "Times New Roman" etc... (DEFAULT: system dependent)

**italic [integer] {lr}**
1 to indicate an italic face, 0 for non-italics. (DEFAULT: 0)

**point_size [integer] {lr}**
Size in points of the text. (DEFAULT: 12)

**template**
Template and Templates classes are used to interface with graphic description files like "graphics.sg2". Templates are created and filed automatically when the templates'read_sg2 method is called. After this, template objects can be assigned to the appearance attribute of many of the graphic and dialog objects in gui.m – thereby linking the contents of the graphic object to the template.

**Class members →**

**name [text]**
This is the name that was saved under when using the SimStudio Icon, Graph or Form editor. It is not the name of the .sg2 file itself, but of an object within the file.

**templates**
The class methods owned by templates can be used to read in ".sg2" files saved by SimStudio.

**find(name) [template reference] {r}**
Searches template_set for a template with the matching name. The template is returned and can be assigned to the appearance attribute of an icon, graph, or form object. A runtime error is generated if the template cannot be found (unless the no_error attribute is non-zero).

**no_error [integer] {lr}**
If <> 0, no runtime error will be generated if the template could not be found after calling find.

*read_sg2({sg2_file_name})*
Given the name of a "sg2" graphics file saved by SimStudio, this method will read the file and adds all templates found inside it to the *template_set*. The file name should include extension. If no path is applied to the name, the current directory is used as the path.

*template_set*
A set of *template* objects that is populated by calling *read_sg2*.

*textfont*
This is a base class for the *linefont* and *systemfont* classes and should not be created directly. Font objects are passed to methods that draw graphic text.

Class members →

*align_horiz*
Horizontal text alignment for both *systemfont* (raster) and *linefont* (vector) style text objects. This attribute should be assigned one of the following constants. (DEFAULT: _left_justified)

  _left_justified
  The horizontal origin of the text is flush with the left side.

  _centered
  The horizontal origin of text is centered.

  _right_justified
  The horizontal origin of the text is flush with the right side.

*align_vert*
Vertical alignment for both *systemfont* (raster) and *linefont* (vector) text. (DEFAULT: _bottom_cell).

  _bottom
  Origin of text at bottom of text line.

  _middle
  Origin of text vertically centered.

  _top
  Origin of text at top of text pixels.

  _bottom_cell
  Origin of text at bottom of text descenders.

  _top_cell
  Origin of text a little above top of largest text characters.
view (view → guiitem)

The purpose of a view object is to act as a container for graphic objects. It defines both a coordinate system and a bounding rectangle for the objects it contains in a set that it owns called graphic_set. A view should be “added” to a window canvas by filing it into the view_set owned by the window. The width and height of the view with respect to the window canvas can be specified. Since the view class is derived from guiitem, the display and erase methods can be used to show or hide all objects in the view_set.

Class members →

graphic_set
Set of objects derived from the graphic class that is to appear within the view. The location_x, location_y and points attributes of these objects must obey the coordinate system defined by set_world.

set_boundaries(xlo, xhi, ylo, yhi)
Sets the size and position of the rectangle containing the world coordinate system within the canvas of the window that it is attached to. NDC (0-32768) coordinates are used.

set_world(left, right, bottom, top)
Sets the end points of the world coordinate system thereby defining a coordinate mapping for graphic objects filed into the graphic_set.

boundary_xlo [integer] {lr}
NDC negative limit along the x axis of the view in the window canvas. (DEFAULT: 0)

boundary_xhi [integer] {lr}
NDC limit along the x axis of the view in the window canvas. (DEFAULT: 32767).

boundary_ylo [integer] {lr}
NDC negative limit along the y axis of the view in the window canvas. (DEFAULT: 0)

boundary_yhi [integer] {lr}
NDC limit along the y axis of the view in the window canvas. (DEFAULT: 32767)

world_xlo [double] {lr}
Minimum X value of world coordinate system. (DEFAULT: 0)

world_xhi [double] {lr}
Maximum X value of world coordinate system. (DEFAULT: 32767.0)
**world_ylo** [double] {lr}
Minimum Y value of world coordinate system. (DEFAULT: 0)

**world_yhi** [double] {lr}
Maximum X value of world coordinate system. (DEFAULT: 32767.0)

**window (window guiitem)**
A *window* object is created for each separate window that is to appear on the screen. Any object that is to appear inside or be attached to the window must be first be filed into one of the sets owned by this object. Subclassing a window and overriding the *action* method allows the application to be notified immediately of user interaction with the scrollbars, or of mouse input events.

**Class members →**

**accept_input(x, y, cursor_style, suspend, animate) yielding (new_x, new_y)**
Waits for the user to click inside a window. A cursor which automatically tracks the movement of the pointer can be shown while waiting for the user to click.

- **_cursor_none**
  Value for the cursor_style argument to accept_input. No cursor will be draw.

- **_cursor_line**
  Value for the cursor_style argument to accept_input. As the mouse moves a line segment is drawn from (x,y) to the current pointer position.

- **_cursor_box**
  Value for the cursor_style argument to accept_input. As the mouse moves a rectangle is drawn anchored at (x,y) to the current pointer position.

- **_cursor_icon**
  Value for the cursor_style argument to accept_input. An icon moves with the mouse.

**action(windowevent) [integer]**
Override this method to receive instant notification of a change to the window by the user. Attributes of the *windowevent* argument contain information about the event that has occurred. The method should return one of the following:

- **_continue**
  Accept the input and continue. If there is a default action (such as closing a window when the user clicks on the "X" in the corner) it will be performed after returning.

- **_block**
  Block the default action (i.e. don't resize or close window).

**color [integer] {lr}**
Assign this attribute to set the background color of the window. Use the `color\rgb` method to obtain a color integer.

### crop_mode [integer]
Sets how the square NDC space is mapped into a rectangular window. The window may be sized to be non-square by the user. There are three choices as to how to map the square canvas coordinates to a rectangular window canvas. (DEFAULT: `_crop_none`) The `crop_mode` property can be assigned to one of the following values:

- **_crop_none**
  Canvas coordinates will occupy the largest centered square within the canvas. All of coordinate system will be visible, but there may be gaps at the ends depending on how tall or wide the window is made.

- **_crop_top**
  The maximum “x” coordinate (32768) is fixed to the right border. The top portion of the canvas coordinates will not be visible if the window is wider than it is tall.

- **_crop_bottom**
  The maximum “y” coordinate (32768) is fixed to the top border. The right portion of the canvas coordinates will not be visible if the window is taller than it is wide.

### form_set
Set of objects derived from `form` such as `dialogbox`, `menubar`, and `palette`. Objects must be filed into this set before they can be displayed.

### get_viewable_size yielding `(width, height)`
Returns the portion of NDC space (0-32767) that is visible given the current size of the window.

### get_view_xy(x_canvas, y_canvas) yielding `(x_view, y_view, view)`
Converts an NDC (x,y) coordinate in the canvas of a window into coordinates defined by the view. The view containing the given canvas coordinates is yielded. This method can be called from the `action` method to convert the given mouse location to coordinates that apply to the selected `view`.

### position_xlo [integer] {lr}
Sets the position of the left edge of the window frame on the screen. (0,0) is the lower left corner and (32767,32767) is the upper right corner of the screen.

### position_xhi [integer] {lr}
Sets the position of the right edge of the window frame on the screen. (0,0) is the lower left corner and (32767,32767) is the upper right corner of the screen.

### position_ylo [integer] {lr}
Sets the position of the bottom edge of the window frame on the screen. (0,0) is the lower left corner and (32767,32767) is the upper right corner of the screen.

`position_yhi [integer] {lr}`
Sets the position of the top edge of the window frame on the screen. (0,0) is the lower left corner and (32767,32767) is the upper right corner of the screen.

`print(use_dialog_flag) [integer] {r}`
Prints the contents of the window. A dialog box containing the printer properties is shown if the "use_dialog_flag" argument is non-zero. Returns 1 if print was successful.

`scrollable_h [integer] {rl}`
Set/get if the window is scrollable in the horizontal direction. If assigned to "1", a horizontal scrollbar is attached to the window. (DEFAULT: 0)

`scrollable_v [integer] {lr}`
Set/get if the window is scrollable in the vertical direction. If assigned to "1", a vertical scrollbar is attached to the window. (DEFAULT: 0)

`status-pane_count [integer] {lr}`
Sets the number of panes in the status bar. A value of zero will erase the status bar. DEFAULT: 0 (no status bar is shown)

`status-pane_text(pane_number) [text] {l}`
Sets the message displayed in the given status pane of the status bar panes. Status panes are numbered starting with "1". (DEFAULT: ").

`status-pane-width(pane_number) [integer]`
Sets the width of the given pane in the status bar. The width is specified in terms of the number of characters the pane can hold. Status panes are numbered starting with "1". DEFAULT: 0

`thumb_pos_h [double] {lr}`
Set/get the position of the thumb from left of scroll bar. If zero, the thumb is at the right of the bar. Max value is 1.0 - thumb_size_h. (DEFAULT: 0)

`thumb_pos_v [double] {lr}`
Set/get the position of the thumb from top of scroll bar. If zero, the thumb is at the top of the bar. Max value is 1.0 - thumb_size_v. (DEFAULT: 0)

`thumb_size_h [double] {lr}`
Set/get the size of the horizontal thumb relative to scrollbar. A thumb with a size of 1.0 will occupy the entire scrollbar. (DEFAULT: 0.1)

`thumb_size_v [double] {lr}`
Set/get the size of the vertical thumb relative to scrollbar. A thumb with a size of 1.0 will occupy the entire scrollbar. (DEFAULT: 0.1)

**windowevent (windowevent → guievent)**
A windowevent object is created by the gui.m runtime library whenever the user performs some action on a window. A *windowevent* instance is passed to the *window'action method.

**Class members →**

- **button_number [integer] {r}**
  Number of the mouse button that was pressed. Mouse buttons are numbered left to right starting with "1".

- **click_count {r}**
  Number of individual clicks for last mouse button down. 1 => single click, 2 => double click, 3 => triple click.

- **modifiers [integer] {r}**
  Can be used to determine if the <shift>, <ctrl> or <alt> keys were being held down when the mouse button was pressed. Holds the ORed product of _shift_down, _ctrl_down and _alt_down.

- **guievent'id [integer] {r}**
  This inherited attribute identifies the type of event.
  - _activate
    Window brought to front by clicking on the header bar or in the canvas.
  - _close
    Attempt to close window by clicking on the X in the top corner of the window frame.
  - _key_down
    Indicates a keyboard event. key_code, key_literal contain the specifics.
  - _key_up,
    Indicates a keyboard event. key_code, key_literal contain the specifics.
  - _mouse_down
    A mouse button was pressed down. (x,y) attributers hold the click location in NDC units (0-32767).
  - _mouse_up
    A mouse button let up. (x,y) attributers hold the click location in ndc units.
  - _mouse_move
    This type of event is given repeatedly as the mouse moves across the canvas. (x,y) attributers hold the click location in NDC units.
  - _reposition
The entire window was dragged to a new location on the screen. (x,y) is new location of window in NDC with (0,0) being the lower left corner and (32767, 32767) being the top right corner of the computer screen.

_reshape
The window was resized by the user. This event may be invoked as the window is being resized or once after the resize operation is complete. (The behavior is system dependent) (x,y) is new size of window in NDC units. This event may also be called when the window is first displayed.

_scroll_x
The thumb of the horizontal scroll bar was moved. (x) is new horizontal thumb position.

_scroll_y
The thumb of the vertical scroll bar was moved. (y) is new vertical thumb position.

_key_code [integer] {r}
Holds a predefined constant that indicates either the value of _literal_key if a alphanumeric key was pressed, or a special key like "_f6_key". Possible values for key_code are: _literal_key, _escape_key, _tab_key, _capslock_key, _shift_key, _ctrl_key, _alt_key, _backspace_key, _enter_key, _insert_key, _delete_key, _home_key, _end_key, _pageup_key, _pagedown_key, _left_key, _right_key, _up_key, _down_key, _numlock_key, _divide_key, _multiply_key, _subtract_key, _add_key, _decimal_key, _clear_key, _keypad0_key, _keypad1_key, _keypad2_key, _keypad3_key, _keypad4_key, _keypad5_key, _keypad6_key, _keypad7_key, _keypad8_key, _keypad9_key, _f1_key, _f2_key, _f3_key, _f4_key, _f5_key, _f6_key, _f7_key, _f8_key, _f9_key, _f10_key, _f11_key, _f12_key, _printscreen_key, _scrolllock_key, _pause_key.

_key_literal [alpha] {r}
For alpha-numeric key press events. This holds the actual key "alpha" value such as "q" or "a". Used only if the key_code attribute is set to _key_literal.

x [double] {r}
X location of mouse click, mouse move, scroll bar thumb position, or window width depending on the event id.

y [double] {r}
Y location of mouse click, mouse move, scroll bar thumb position, or window height depending on the event id.

### 6.2 3d.m Objects

**3dcamera (3dcamera → 3dnode)**

A 3dcamera is used show a view of the 3dworld on the canvas of a window. For each 3d.m application, at least one camera must be created, oriented, and filed in the
“camera_set” owned by the 3dworld, otherwise nothing will be seen. Since the 3dcamera object is derived from 3dnode, it inherits the set_location, set_orientation, and set_forward methods. All can be used to position and orient the camera much in the same way that a “real” camera would be positioned.

A 3dcamera can optionally be filed into a node_set allowing it to move and rotate automatically with the owner node. For example, a 3dcamera could be filed into the node_set owned by a jet plane and if positioned at the cockpit, would provide the same shifting view of scenery that a pilot would see as the plane flies.

The portion of the 3dworld seen by the camera is projected onto a rectangular area of the canvas called a viewport. Normally, the viewport dimensions will match the canvas dimensions. In fact, if the viewport_autosize attribute of 3dcamera is non-zero (default) the view will automatically size to match the canvas whenever the user resizes the window. However, if more than one camera is present, the multiple views should be mapped to different regions of the canvas. The set_viewport method specifies the viewport rectangle (in pixels) with the (0,0) coordinate located at the lower left corner of the canvas. The call to set_viewport will usually have to be made in response to the user resizing the window. (At this time, the new pixel dimensions of the canvas are known, allowing the viewport to be sized appropriately.) This can be accomplished by subclassing the 3dwindow object and overriding the “action” method. If the event id is “_resize” then the “x” and “y” attributes of the event will have the new size of the window.

```
define set_viewport as a method given
  2 integer arguments,  "x, y in pixels.
  2 integer arguments,  "width, height pixels.
  1 integer argument   "1=>size viewport to window canvas

  "Defines the viewport within the window that will display what is seen
  "by the Camera. Values are given in pixels with (0,0) located at the
  "lower left corner of the window.
  "DEFAULT: <viewport_autosize = 1>
```

Two different types of projections are possible when using a 3dcamera, orthographic, and perspective. When using orthographic projections, the distance from the camera to the viewed object does not affect the size of its image seen on the canvas. A simple view volume defines to portion of the 3d world that is “seen” by the camera. This box is of course oriented and positioned with respect to the camera, not the world. The set_orthographic method specifies the view volume. If the “auto resize” flag is specified, the view volume will be adjusted automatically when the user resizes the window. Values are specified relative to the camera’s own orientation and location.

```
define set_orthographic as a method given
  2 double arguments,  "left, right,
  2 double arguments,  "bottom, top,
  2 double arguments,  "far, near,
  1 integer argument   "1=> adjust size automatically after
                     "window resize

  "sets up an orthogonal (parallel) projection. It is assumed that the
```
'eye is located at the "location" of the camera and looking down the
'negative z axis. The ortho_xlo, ortho_xhi, ortho_ylo, ortho_yhi,
'ortho_zlo, ortho_zhi, and ortho_autosize attributes are set.

When using a perspective projection, distant objects will appear smaller. This form of
projection provides a more realistic view. The set_perspective method is used to specify
the attributes of a perspective projection. The distance from the camera to both the near
and far clipping planes is specified as well as the angle (in degrees) of the view from the
location of the camera. Decreasing the angle of view has the same effect as “zooming in”
with a traditional camera.

define set_perspective as a method given
  1 double argument,     ''angle > 0.0, < 180.0 degrees
  1 double argument,     ''aspect ratio.
  2 double arguments,    ''near, far clipping planes
  1 integer argument     ''1=>compute aspect ratio after win resize
'sets up a perspective viewing transformation. The view angle is
'relative to the eye. The aspect ratio will equal the width of the
'projection divided by its height. The location of clipping planes is
'specified relative to the camera. Geometry in front of the near, or
'behind the far clipping planes is clipped.

By default, a camera will use the perspective projection.

A camera can be programmed to track (point at) a particular 3dnode object that is filed in
the same world_set. Assigning the tracked_node attribute to a 3dnode instance will cause
the camera to track the node automatically, even if both objects are moving. The camera
will remain oriented so that the global y-axis appears to point up. Tracking can be
stopped by assigning tracked_node to zero.

define tracked_node as a 3dnode reference variable
  monitored on the left
'Can be assigned to a 3dnode in the same 3dworld. Will cause the
'camera to automatically point to the given node. Must be assigned to
'zero before the tracked_node can be destroyed.

3devent (3devent → gui.m:guievent)
An instance of a 3devent is passed to the 3dwindow'action method when the user types
on the keyboard, clicks in the window with the mouse, or resizes the window. 3devent is
derived from gui.m’guievent and each instance will contain attributes pertinent to the
type of event.

button_number  ''number of the mouse button that was pressed
click_count    ''1 => single click, 2 => double click
key_code       ''either "_literal_key" or a special key like "_f6_key"
key_literal    ''letter or number that was pressed
modifiers      ''presence of alt, ctrl and/or shift keys during event
x,y            ''location of mouse click, mouse move, new canvas size
  (pixels)
The **id** attribute contains an enumerated constant describing what type of event had occurred. The following events are currently handled:

- _activate, "'window frame brought to front"
- _close, "'attempt to close window"
- _key_down, "'key pushed down. key_code, key_literal contain info"
- _key_up, "'key released. key_code, key_literal contain key info"
- _mouse_down, "'(x,y) is click location in pixels"
- _mouse_up, "'(x,y) is click location in pixels"
- _mouse_move, "'(x,y) is new mouse location in pixels"
- _mouse_wheel_forward, "'spin mouse wheel away from user"
- _mouse_wheel_back, "'spin mouse wheel toward user"
- _reposition, "'(x,y) is new location of window in pixels"
- _resize, "'(x,y) is new size of window in pixels"

The user may choose to hold down the **shift**, **ctlr**, or **alt** key during the event. This can be detected by examining the **modifiers** attribute for the presence of the _alt_mod, _ctrl_mod, _shift_mod bits.

If an application needs to respond to selection by mouse of a visible 3dnode, the **3dworld.select_node** method can be called to see if a node in the world has been clicked. The 3dnode object defines an “action” method that can be overridden by classes that receive mouse clicks or other events. An example of a user defined “action” method is shown below. This method responds to “shift” clicks on a node owned by “the_world”.

**Method my_3dwindow'action(elevent)**  
Define node as a 3dnode reference variable  

If id(elevent) = 3devent'mouse_down and  "'mouse click  
and.f(modifiers(elevent), 3devent'shift_mod) <> 0  "'shift key down  
Let node = select_node(the_world)(x(elevent), y(elevent))  
If node <> 0  
Call action(node)(elevent)  "'pass event to 3dnode'action  
Always  
Always  
End

The 3devent instance will be destroyed after the **3dwindow'action** method is called.

**3dfaces (3dfaces → 3dgraphic → 3dnode)**

This class can be used to draw various segmented surfaces. Each “face” must be planar but exists in 3d space. Coordinate geometry is defined by a 1-dim double array attribute called **points**. The points are assigned to the array with successive array elements assigned to x then y then z coordinate values for each point. I.e. (x1, y1, z1, x2, y2, z2, x3, y3, z3 ...) The size of the array must therefore be three times the number of points (npoints * 3). A 1-dim real array attribute called **normals** can be assigned to specify a normal vector corresponding to each point. The size of **normals** array must match the points array. The **format** attribute can be assigned to one of the following enumerated constants:
_triangles:
   Each successive group of 3 points defines a triangle
_triangle_strip:
   For each index n > 2 a triangle is defined by points at n-2, n-1 and n in counter-clockwise order. Sometimes this is called the “triangular mesh” or just “mesh”.
_triangle_fan:
   For each index n > 2 a triangle is defined by points at n, n-1 and 1
_quads:
   Each successive group of 4 points defines a quadrilateral face.
_quad_strip:
   For each 2 indices n, n-1, a new quadrilateral is composed of points at n-3, n-1, n, n-2 in counter-clockwise order.
_polygon:
   The outline of the polygon is defined by points. Again, the points for front facing polygons should be arranged in a counter-clockwise order.

The diagrams below show how the points array defines specifically formatted geometry

![Diagrams showing 3D graphics triangles format](image)

**Figure 55: 3D graphics' triangles format**
Figure 56: the 3dgraphic\_triangle\_strip format.

Each successive point defines a new triangle.

Points:
- points(1) 'x1
- points(2) 'y1
- points(3) 'z1
- points(4) 'x2
- points(5) 'y2
- points(6) 'z2
- points(7) 'x3
- points(8) 'y3
- points(9) 'z3
- points(10) 'x4
- points(11) 'y4
- points(12) 'z4
- points(13) 'x5
- points(14) 'y5
- points(15) 'z5
- points(16) 'x6
- points(17) 'y6
- points(18) 'z6

Figure 57: the 3dgraphic\_triangle\_fan format.

Each successive point defines a new triangle using the first "center" point.

Points:
- points(1) 'x1
- points(2) 'y1
- points(3) 'z1
- points(4) 'x2
- points(5) 'y2
- points(6) 'z2
- points(7) 'x3
- points(8) 'y3
- points(9) 'z3
- points(10) 'x4
- points(11) 'y4
- points(12) 'z4
Figure 58: 3dgraphic'_quads format.

Figure 59: 3dgraphic'_quad_strip format.
Figure 60: 3dgraphic’_polygon format.

The material attribute inherited by 3dnode can be set to a 3dmaterial instance. The material colors will be applied to each face. The texture_points real array attribute will be utilized if the texture_name attribute of the material has been specified. (See the reference for the 3dMaterial class for a description of supported texture mapping functions.)

To apply a different color to each vertex, the colors array attribute can be assigned which has the color values which will be applied to each vertex. Intensities are represented with reals in range [0.0,1.0] ordered as: R1, G1, B1, R2, G2, B2, ... OR in the order R1, G1, B1, A1, R2, G2, B2, A2, ... if "alpha" values are used. The size of the colors array must be 3 * the number of vertices for {RGB} values and 4 * the number of vertices for {RGBA} values.

The edge_flags array attribute of 3dfaces can be assigned to a 1-dim integer array sized to match the points array. Each element value defines the visibility of the corresponding face edge (with a non-zero value meaning “visible”). If the array is not assigned, all edges are visible.

In some cases, a performance improvement can be gained by using indexed vertex arrays. If vertices are naturally shared in the geometry, the unique vertices can be specified in the points array with index values of (possibly multiple) references stored in the indices array attribute. Using this method can save memory and improve performance. For example, a cube consists of 8 unique vertices. Using the _quads format without indexed vertex arrays to define the faces, requires 24 vertices, or a points array of dim (24 x 3) = 72 (576 bytes). Using vertex arrays, the smaller indices integer array would contain the 24 elements to define zero based indices into the points array (96 bytes). The points array would contain only the unique eight vertices, or (8 x 3) = 24 elements (192 bytes). The total memory requirement for the geometry is (192+96) = 288 bytes for index vertex arrays, versus 576 bytes. See below:
Figure 61: Using the 3dfaces'indices attribute to define a cube.

Three helper methods called set_point, set_normal, and set_texture_point can be used to assign values in the points, normals, or texture_points arrays respectfully. The (1 based) index is given followed by the vector or coordinates. The corresponding array must be reserved beforehand and must be large enough to store the vector at the specified index.

define set_normal as a method given
1 integer argument, ''index
3 real arguments ''nx,ny,nz normal vector
''This helper method assigns one of the normal vectors in the "normals" array to (nx,ny,nz).

define set_point as a method given
1 integer argument, ''index
3 double arguments ''px,py,pz coordinates of vertex
''This helper method assigns a point (vertex) in the 'points' array to (px,py,pz).

define set_texture_point as a method given
1 integer argument, ''index
2 real arguments ''(s,t)
''This helper method assigns a texture coordinate in the 'texture_points' array to (s,t).

Another helper method called compute_normal_vectors can be called to automatically assign normal vectors to the normals array based on the geometry in the points array and the format attribute. This method does not automatically smooth the surface and will result in a "faceted" look. The normals arrays must be reserved and the points and format attributes assigned before calling compute_normal_vectors.

3dgraphic (3dgraphic → 3dnode)
The 3dgraphic is the base class for all objects whose geometry is defined by program code, and not by offline models stored in a file. Classes derived from 3dgraphic can be seen by a 3dcamera and illuminated by a 3dlight. Since 3dgraphic is derived from 3dnode, instances must be filed into a node_set (owned by either the 3dworld or by another 3dnode attached to a 3dworld).

Geometry and surface materials can be specified by sub-classing 3dgraphic and overriding the draw method. The draw method is called automatically when the system
needs to have the 3dgraphic object rendered. (Calling the \textit{3dwindow\textquotesingle display} method will invoke this method. \textit{Draw} may also be called as a result of event handling). The \textit{draw} method is programmed to make calls to \textit{begin\_drawing} and \textit{end\_drawing} class methods to define each surface. After the call to \textit{begin\_drawing}, calls to class methods such as \textit{draw normal}, \textit{draw texture\_coordinate}, and \textit{draw vertex} allow geometry to be specified for the surface. A call to \textit{end\_drawing} will mark the end of the geometry description started by \textit{begin\_drawing}.

\begin{verbatim}
define begin\_drawing as a method given
   1 integer argument  'format

define end\_drawing as a method

The “format” argument to \textit{begin\_drawing} must be one of the following predefined constants:

\_points:
   Draw dots in 3d space. Each call to \textit{draw\_vertex} adds a dot. (See \textit{3dpoints} class).

\_lines:
   Each 2 successive vertices defines a line segment (see \textit{3dlines} class).

\_line\_loop:
   Each vertex is connected to the next with the first connected to last (see \textit{3dlines} class).

\_line\_strip:
   Each vertex is connected to the next (see \textit{3dlines} class).

\_triangles:
   Each successive group of 3 vertices defines a triangle

\_triangle\_strip:
   For each index \( n > 2 \) a triangle is defined by vertex at \( n-2, n-1 \) and \( n \) in counter-clockwise order. Sometimes this is called the “triangular mesh” or just “mesh”.

\_triangle\_fan:
   For each index \( n > 2 \) a triangle is defined by vertex at \( n, n-1 \) and 1

\_quads:
   Each successive group of 4 vertices defines a quadrilateral face.

\_quad\_strip:
   For each 2 indices \( n, n-1 \), a new quadrilateral is composed of vertices at \( n-3, n-1, n, n-2 \) in counter-clockwise order.

\_polygon:
   The outline of the polygon is defined by vertices. Again, the vertices for front facing polygons should be arranged in a counter-clockwise order.

The \textit{draw\_normal} class method can be called to specify a normal vector to be applied to all subsequent vertices. \( X, y, z \) components are specified and the given vector should be normalized to the range \((0.0, 1.0)\).

\begin{verbatim}
define draw\_normal as a method given
   3 double arguments  ''(x,y,z)
\end{verbatim}
\end{verbatim}
The \texttt{draw\_texture\_coordinate} method can be called to map the next vertex to a given 2d texture coordinate. These “texture” coordinates reference the texture of the current material (see \texttt{3dgraphic\_set\_material}). The texture is a bitmapped image with the (0.0, 0.0) coordinate in the lower left corner and (1.0,1.0) in the upper right. These two “s” and “t” coordinates are the given arguments to the method.

\texttt{define draw\_texture\_coordinate as a method given 2 double arguments \'(s,t)\}

The \texttt{draw\_vertex} class method adds a new 3d vertex to the current drawing given the x, y, and z components.

\texttt{define draw\_vertex as a method given 3 double arguments \'(x,y,z)\}

The \texttt{draw\_vertex}, \texttt{draw\_normal}, and \texttt{draw\_texture\_coordinate} methods MUST be called between \texttt{begin\_drawing} and \texttt{end\_drawing} methods. Other class methods can be called to set features for drawing materials and lines. The \texttt{set\_color} method can be used to define surface color or, if called before \texttt{draw\_vertex}, can set the color applied to individual vertices.

\texttt{define set\_color as a method given 1 integer argument, \'_front, _back, or _front\_and\_back\' 1 integer argument, \'_ambient, _diffuse, _specular\' 1 integer \'color\' argument \'color (color\_rgb(r,g,b))\}

The first argument indicates which side the color should be applied to, and must be \_front, \_back or \_front\_and\_back. The second gives the type of lighting, \_ambient, \_diffuse, or \_specular that should reflect the color (see \texttt{3dlight}). The third argument is the color value obtained from the \texttt{gui.m:color\_rgb} class.

Attributes of the \texttt{3dmaterial} object (colors and texture) can be applied to drawings as well. The \texttt{set\_material} method can be called before \texttt{begin\_draw} and takes a \texttt{3dmaterial} instance as an argument. This instance must be filed in the \texttt{3dworld\_material\_set} before the \texttt{draw} is invoked.

\texttt{define set\_material as a method given 1 3dmaterial reference argument \'pointer to material\}

When lines are being drawn, the \texttt{set\_pen\_size} and \texttt{set\_pen\_pattern} methods can be called to set the line width and pattern.

\texttt{define set\_pen\_size as a method given 1 integer argument \'line width in pixels\}

\texttt{define set\_pen\_pattern as a method given 1 integer argument \'line style\}
These methods must be called before begin_drawing. The current pen size is always in pixels. The pattern is one of the following constants: _solid, _long_dash, _dotted, _dash_dotted, _medium_dash, _dash_dot_dotted, _short_dash, _alternate. (See 3dlines for more).

The shininess of the surface (used for specular lighting) is set by calling the set_shininess class method.

```
define set_shininess as a method given
  1 integer argument,   
    ''_front, _back, or _front_and_back
  1 double argument     
    ''shininess (0.0 to 1.0)
```

The first argument must be one of the constants _front, _back, or _front_and_back. The second “shininess” argument must be a value between 0.0 and 1.0 (see 3dlight).

By default, both the front and back sides of a surface are visible. In some cases, back or front side of an object is never seen. Faster rendering can be achieved for single sided drawings by defining which side is to be made visible. The set_visibility method can be called to accomplish this. The method call must be made before begin_drawing.

```
define set_visibility as a method given
  1 integer argument  
    ''_front, _back or _front_and_back
```

Individual edges for surfaces specified by calls to draw_vertex are normally visible. For some edges such as those on the interior of a complicated shape, this visibility can lead to unwanted artifacts when the 3dgraphic is displayed. The set_edge_visibility method can be used to hide or show future edges added to the shape by draw_vertex.

```
define set_edge_visibility as a method given
  1 integer ''Boolean'' argument  
    ''0=>invisible, 1=>visible 
    ''specifies visibility of future edges of a face
```

By default the draw method is only called once by the system when the 3dgraphic first appears. If the geometry or surface has changed, the update_drawing method must be used to indicate that the 3dgraphic is “old” and that the draw method should be invoked (the draw method should never be called directly).

```
define update_drawing as a method given
  1 integer argument   
    ''_always, _once, _never
    _never
      Draw will not be called. Graphic will effectively be hidden from view.
    _once
      Draw called the next time the window is refreshed. The image of the 3dgraphic will be “cached” for future refreshes of the screen. The update_drawing method must be called for draw to be invoked again.
    _always
      Draw will be called each time the canvas is refreshed. This is useful for objects with constantly changing geometry.
3dlight ( 3dlight → 3dnode)

A 3dlight object represents a light source in 3d space. It is derived from 3dnode and inherits its location, orientation and “set” properties. Light sources are apparent only if the lighting attribute of the 3dworld is non-zero. A single light source can have “ambient”, “diffuse” and “spot” properties.

Ambient

This type of light is non-directional and non-positional (location and orientation is ignored). Ambient light will illuminate all surfaces ignoring normal vectors. The ambient term is used simply to keep shadows from turning pitch black. The relative intensity of ambient light can be set by assigning the ambient_color attribute to an rgb value returned from the gui.m:color rgb class method. If less ambient light is required, darker colors should be used. For example, setting ambient_color to color’rgb(0.25, 0.25, 0.25) will provide a 1/4th intensity of ambient light.

Diffuse

Diffuse light is reflected from a surface in all directions. The amount of reflected light is determined by the “diffuse” color of surface material (see 3dmaterial). Rough surfaces should have a relatively bright diffuse_color attribute. Both diffuse and specular light allows objects to be shaded based on surface normal vectors. Surface elements with normals pointing at the light source will be illuminated while surfaces with normals orthogonal to the light source’s direction will not. The diffuse_color attribute determines the color and intensity of diffuse light.

Specular

Specular light is reflected from a surface in mostly one direction. Shiny surfaces reflect more of the specular light than rough surfaces. The amount of reflected light is determined by the “specular” color of surface material (see 3dmaterial). Surface elements with normals pointing at the light source will be illuminated while surfaces with normals orthogonal to the light source’s direction will not. The specular_color attribute determines the color and intensity of this light.

The variety attribute defines the type of lighting. One of the following constants can be assigned.

_positional

Light emanates in all directions from a single position. This position is the location attribute inherited from 3dnode.

directional

Light travels in only a single direction which is determined by the “forward” vector inherited form 3dnode. The location attribute is ignored.
Spot lighting is both positional and directional. The “location” attribute determines spot light position while orientation (i.e. the forward vector) inherited from 3dnode determines its direction. The intensity of spot lighting is concentrated along the forward vector’s direction. The spot_cutoff attribute is an angle (in degrees) that indicates how the light fans out from its source location. The default value is 45 degrees.

The spot_cutoff attribute specifies the radial angle (in degrees) of the cone of light emanating from a spot variety light. The angle is specified from the center line to the edge of the cone. This attribute must range from 0 to 90.

3dlines ( 3dlines → 3dgraphic → 3dnode)
This class derived from 3dgraphic can be used to draw various segmented lines. Coordinate geometry is defined by a 1-dim double array attribute called points. Vertex data is packed into the points array as x1, y1, z1, x2, y2, z2, x3, y3, z3, .... The format attribute can be assigned to one of the following enumerated constants:

- **_lines:** Each successive group of 2 points defines a separate line segment (see Figure 62).
- **_line_strip:** Successive points are connected by line segments. The first and last points are not connected (see Figure 63).
- **_line_loop:** Successive points are connected by line segments. The first and last points are connected (see Figure 64).

The diagrams below show how the points array defines specifically formatted geometry:

![Figure 62: 3dgraphic'_lines format](image)
Other attributes of the 3DLines object include the color, width and pattern. The color can be obtained from the gui.m:color class, and applies to ambient, diffuse and specular colors. Line width is specified in pixels. The line pattern refers to the dash style found in the 3dgraphic class and must be one of the following:

```
define _solid=0, _long_dash, _dotted, _dash_dotted, _medium_dash, _dash_dot_dotted, _short_dash, _alternate as constants
```

In most cases the 3dlines object will be used for illustration purposes and the application will not want it to be shaded. If this is the case, the 3dnode:enabled(_lighting) can be assigned to “0”. This will cause the lines to be shown regardless of how lights are positioned.
3dmaterial

The 3dmaterial object defines the "skin" of the object being drawn. It is not derived from 3dnode but instead can be set as the current “drawing” material for the 3dgraphic’draw method. Objects derived from 3dshape also have a material attribute that can be assigned an instance of a 3dmaterial object.

A 3dmaterial can be used to define reflectivity a surface has to diffuse, ambient, and specular colored light (see 3dLight). The diffuse_color, ambient_color, and specular_color attributes are used for this purpose.

define ambient_color, diffuse_color, specular_color
    as integer ''color reference'' variables monitored on the left
    ''ambient_color DEFAULT: color'_white
    ''diffuse_color DEFAULT: color'_white
    ''specular_color DEFAULT: color'_black

For specular light the shininess attribute relates to the “specular exponent” of the surface. This value must range from 0 to 1. Higher values lead to smaller, sharper highlights, whereas lower values result in large and soft highlights. If the surface is to be shiny (i.e. metallic in nature) both the specular_color and shininess attributes should be large.

Texture mapping is also supported through the 3dMaterial class. Texture mapping allows a 2d pixel image (such as a windows .BMP file) to be plastered onto a 3d surface. The mapping of 3d geometry to 2d points in the image file is done using texture coordinates (see 3dgraphic’draw_texture_coordinate). Basically, when draw_texture_coordinate is called before draw_vertex, the 2d texture coordinate is mapped or “attached” to the 3d vertex. The result is a smooth texturing of the 2d image over the surface.

The set_texture_data method is used to assign the data used for the texture.

define set_texture_data as a method given
    1 pointer argument, 'array of texture data
    1 integer argument, 'texture data format constant
    1 integer argument, 'texture data mode constant
    2 integer arguments, 'width, height (must be power of 2, - 256, 512
    1 2-dim real argument'color map if data format is to be indexed
    'size is (3,4) by n where n is a power of 2

The first argument points to either the texture color data itself, or an array of indices into a color map provided in the last argument. The 2nd and 3rd arguments define many different combinations of data formatting as shown in the following table:

<table>
<thead>
<tr>
<th>Texture data (arg 1)</th>
<th>Format (arg 2)</th>
<th>Mode (arg 3)</th>
<th>Map (arg 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit integer {rgb} elements</td>
<td>rgb_format</td>
<td>integer1 mode</td>
<td>0</td>
</tr>
<tr>
<td>16-bit integer { rgb } elements</td>
<td>rgb_format</td>
<td>integer2 mode</td>
<td>0</td>
</tr>
<tr>
<td>32-bit real { rgb } elements</td>
<td>rgb_format</td>
<td>real mode</td>
<td>0</td>
</tr>
<tr>
<td>8-bit integer {rgba} elements</td>
<td>rgba_format</td>
<td>integer1 mode</td>
<td>0</td>
</tr>
<tr>
<td>16-bit integer {rgba} elements</td>
<td>rgba_format</td>
<td>integer2 mode</td>
<td>0</td>
</tr>
</tbody>
</table>
If the last argument (color map) is zero, the color data is found directly in the first argument in row major order. If the “color map” argument is non-zero, the “texture data” argument contains an array of indices into this table. NOTE that the color map is always 32-bit real and can be stored as {rgb} or {rgba} format depending on the 2nd (data format) argument.

An easier way to use texture mapping is to utilize the image in a 2d graphics file. The texture_name attribute of 3dMaterial can be assigned the name of the file containing the image. TARGA graphic (.tga) files and Window Bitmap (.bmp) files are supported. (JPEG files can be converted to BMP by a variety of windows programs). The width and height of images should be a power of 2, for example 128 by 256, 16 by 64, 512 by 32, etc. If a texture file name is assigned, texture data defined by the set_texture_data method is ignored.

The 2d coordinates for a texture image range from 0.0 to 1.0. Coordinate are defined by an s axis and a t axis. The s-axis is horizontal and the t-axis is vertical with (s=0.0, t=0.0) located at the lower left corner of the image and (s=1.0, t=1.0) located at the upper right corner.

For texture coordinate values greater than 1.0 or less than 0.0 the mapping will be handled based on the values of the texture_wrap_s and the texture_wrap_t attributes. When an attribute is set to _repeat, the pixel image is repeated as (s,t) values increase past 1.0 (or decrease past 0.0). If the texture wrapping attributes are set to _clamp_to_edge, the same pixel values found at [s,t] = 1.0 will be copied for all values of [s,t] > 1.0. Pixel values found at [s,t] = 0.0 will be repeated for [s,t] < 0.0.

In some cases it is necessary to specify which sides of a surface can be made visible. If the front or back side is always hidden, some performance improvement can be made by marking that side as such. The visibility attribute of 3dMaterial controls which sides are visible (an invisible side will appear translucent when facing the 3dcamera).

define visibility as an integer variable monitored on the left
One of the constants 3dmaterial’_front, 3dmaterial’_back, or 3dmaterial’_front_and_back can be assigned to the visibility attribute (3dmaterial’_front_and_back is the default value). The 3dmaterial’_front side is defined by a counterclockwise winding of the vertices. The “right-hand thumb” rule can be used as mnemonic reminder. If a fist is made with the right hand and the vertices of a polygon are ordered in the direction the fingers point, then the thumb points out from the “front” of the surface.

A caveat to using the 3dmaterial object is that each instance must be filed in a material_set before it is rendered. 3dmaterials should be filed into the set owned by the 3dworld in which it will be visible. The 3dmodel also owns a material_set containing materials to be used within the model.

3dmodel
The 3d surfaces and geometry shown in the 3dworld can be defined in one of two ways. The 3dgraphic and its subclasses allow the application to specify the geometry and materials and runtime. The 3dmodel class allows the surfaces and materials to be loaded from a file. Basically, a single instance of a 3dmodel is created for each separate 3d file. The read method loads the contents of the file creating surfaces and materials, which are saved in memory.

```
define read as a method given
1 text argument, ''file name including .3ds, .dxf, .sg2 extension
1 text argument     ''name of model in the file (.sg2 files only)
```

The first argument specifies the name of the file. Currently, the file must be in either autodesk 3dStudio or “.3ds” format (the extension is required), AutoCAD dxf format, or SIMGRAPHICS II “.sg2” format. For .sg2 files, the name of the model within the file is provided in the second argument.

The 3dmodel class is not derived from 3dnode and therefore cannot be shown in a window directly. The 3dmodel can be assigned to the model attribute of a 3dnode. This will provide a link from the image of the 3dmodel to the 3dnode. Many instances of 3dnode can reference the same 3dmodel instance. This scheme allows a single model to be drawn in different locations and orientations in the world.

Models may be designed in the 3d editor to have well defined components. For example, a ‘tank’ model may have a “turret” component that can be rotated with respect to the tank. The turret sub-component may in turn have a ‘gun’ sub-component. The application may need access to these sub-components at runtime (i.e. rotate the turret, move the gun in and out when it fires, etc). If sub-components like this are defined in the graphics editor, they will be preserved when model is loaded in the application. The node_set owned by the 3dmodel will contain these components. The find method can be called to perform a depth-first search for a sub-component, provided that the component has been given a name in the graphics editor.

```
define find as a 3dnode reference method given
```
Suppose the application wanted to represent the turret sub-component using an object derived from 3dnode (instead of the 3dnode base class). In this case the 3dmodel would be sub-classed and its create_component method overridden.

```
define create_component as a 'virtual' 3dnode reference method given
  1 text argument, 'name of the component
  1 text argument, 'name of required class

  'This method is called during the execution of the "read" method to create a new sub-node component. The name given in the model file is provided in argument 1. By default this method will create an instance of the class named by argument 2, but can be overridden to create a sub-class of arg 2.

Create_component is called automatically for each separate component that is created at the time a 3dmodel is read. In the above example, create_component would be implemented to create and return a "turret" object if the first argument specified the name assigned to the turret component in the graphics editor. See below.

method tank_model'create_component(component_name, class_name)
  define my_turret as a turret reference variable
  if component_name = "Turret_for_tank"
    if class_name <> "3dnode" ' 'for safety, check base class
      write as "error, 3dnode expected", /
      stop
    always
    create a turret called my_turret
    return with my_turret
  otherwise
  return with 3dmodel'create_component(component_name, class_name)
end
```

In some cases, the application may require individual images of a 3dmodel to appear differently. For example, if many “tanks” were to be displayed, each turret will have a different orientation. In this case, the 3dnode'load method would be called. This method will make copies of each sub-component and place the nodes into the appropriate node_set.

In the following code, two tanks are loaded from a common model, read from the file “tank.3ds”. The turret on the first tank is rotated 90 degrees, while the second turret is rotated –45 degrees.

Define tank1, tank2, as tank reference variables
Define the_model as a tank_model reference variable

```
Create tank1, tank2, the_model
File the_model in model_set(the_world)
File tank1 in node_set(the_world)
File tank2 in node_set(the_world)
```
Call read(the_model)("tank.3ds", "") ' 'create_component called
Call load(tank1)(the_model) ' 'copy_attributes called
Call load(tank2)(the_model)
Call rotate_y(find(tank1)("Turret_for_tank"))(pi.c / 2.0)
Call rotate_y(find(tank2)("Turret_for_tank"))(-pi.c / 4.0)

In the above example, the “load” method call makes copies of all components in the model. Individual attributes are copied by an internal call to the copy_attributes method. The sub-class can override this method if it defines attributes that need to be copied with the rest of the components. In our tank example, suppose the “turret” class defines attributes “azimuth” and “attitude” that are initializes in the model. We want these values to be propagated when turret is copied (via the load method).

Begin class turret
    Every turret is a 3dnode and has
    A azimuth,
    An attitude, and
    Overrides the copy_attributes
    Define azimuth, attitude as double variables
End

The copy_attributes method’s implementation would look like this:

method turret'copy_attributes(node)
    define p as a pointer variable
    define original_turret as a turret reference variable

    let p = node ' 'necessary due to original prototyping
    let original_turret = p

    let azimuth = azimuth(original_turret)
    let attitude = attitude(original_turret)

    call 3dshape'copy_attributes(node)
end

There are a couple more options regarding the 3dmodel that can be set before the model is read from the file. They are _smoothing, and _cache_model. The options are set via a left handed use of the enabled method. For example:

define the_model as a 3dmodel reference variable

... let enabled(the_model)(3dmodel'_smoothing) = 1 ' 'turn on smoothing
let enabled(the_model)(3dmodel'_smoothing) = 0 ' 'turn off smoothing

The _smoothing option will cause normal vectors to be recomputed at the time the model is read from the file. This will give a smooth appearance of the model. It is off (0) by default.
The `_cache_model` option, if on, will cause the runtime library to create an internal “call list” for the model at the time a 3dnode which references it (via the _model_ attribute) is first drawn. For the case of multiple 3dnode objects referencing the same static model, this will improve performance. This option is not relevant if the _load_ method is used to link the 3dnode to the model.

After a model is read, the size of the model can be determined if necessary. The _get_bounding_box_ method can be called to dimensions of a model in the x, y and z directions.

```plaintext
define get_bounding_box as a method yielding
    3 double arguments, ''(xlo,ylo,zlo)
    3 double arguments ''(xhi,yhi,zhi)
''Computes the smallest 3d box that will enclose the model
''Must be called after "read" to be effective.
```

Instances of 3dmodel must be filed into the _model_set_ owned by the 3dworld in which the model is to appear. This must be done before the window is displayed. Materials used in the model are automatically filed into the _material_set_ owned by 3dmodel when the 3dmodel’s _read_ method is called.

### 3dnode

The 3dnode object is the base class of all objects that can appear in a 3dworld. Nodes are used to build the scene-graph. This is a directed acyclic graph that represents the spatial relationship between objects. A node owns a set of nodes called _node_set_ and may also belong to a _node_set_ allowing a hierarchy to be built. When the location or orientation of a 3dnode is changed, all 3dnode instances attached to the scene-graph via the _node_set_ are repositioned as well. In order to be made visible, a 3dnode must be attached to a scene-graph. The scene-graph is rooted at the 3dworld object, which also owns a _node_set_. Visible nodes must therefore be filed in a _node_set_ owned by either another 3dnode or a 3dworld. See Figure 65.
An instance of a 3dnode can therefore be used to create groups of objects; Moving or rotating the 3dnode representing the “group” will move/rotate all nodes in the group. Complex objects can be divided into “sub-nodes”. For example, a 3d tank may contain a turret sub-node which in turn contains a “cannon” node. Moving the tank will also move both the turret and gun. Rotating the turret will also rotate the cannon.

Before a 3dnode can be destroyed, it must first be removed from any owner set. Destroying a 3dnode will automatically destroy nodes contained in its node_set. To prevent this from happening, nodes should be removed from the set prior to executing the destroy statement.

Properties of the 3dnode are its location and orientation. The location_x, location_y, and location_z attributes represent the x, y, and z coordinates of the node’s position with respect to its owner node in the scene-graph. Right handed use of these attributes is allowed, but the set_location method should be used to set these attributes.

```
define location_x, location_y, location_z as double methods
define set_location as a method given
    3 double arguments "x, y, z"
```

The orientation is defined by 2 vectors, “forward” and “up” (See Figure 66). The forward vector indicated the direction of the local z-axis relative to the owner node in the scene-graph. By default, the forward vector is (0.0, 0.0, 1.0) which would point a node in the same direction as its owner node. The “up” vector is the local y-axis relative to the node’s owner in the scene_graph. By default, this is (0.0, 1.0, 0.0). The local x-axis is computed automatically by taking the cross product of these vectors.
The attributes forward_x, forward_y, forward_z, up_x, up_y and up_z can be used on the right. Both vectors must always be normalized and orthogonal to each other, so they should be assigned at the same time. The set_orientation method allows both the forward and up vectors to be updated.

```
define forward_x, forward_y, forward_z as double methods
define up_x, up_y, up_z as double methods
define set_orientation as a method given
    3 double arguments, "forward_x, forward_y, forward_z"
    3 double arguments "up_x, up_y, up_z"
```

Another method called set_forward allows the forward direction to be specified alone, while the “up” vector is computed automatically. This vector is computed in such a way that its projection onto the positive y-axis is maximized. (for 3dcameras, this will prevent the view from tilting assuming the “floor” of the scene lies in the x-z plane)

```
define set_forward as a method given
    3 double arguments "forward_x, forward_y, forward_z"
```

Another method that may be useful for setting the orientation of a 3dnode is the aim method. Calling the aim method will set the forward direction of the node so that it “points at” a given location. The location should be in global coordinates; non-local to the 3dnode. (The 3dworld’get_location method can be used to convert a location from local to global coordinates). When aim is used on a sub-component, the (local) orientation (forward and up vectors) of the sub-component will be modified so that the sub-component points (with its positive z-axis) at the given location. The aim method can only be called after the 3dnode has been filed into the node_set. Since the aim method uses the orientation and location of parent nodes, it should be called after all the location of all parent (grand-parent, etc.) nodes have been initialized.

```
define aim as a method given
    3 double arguments "target_x, target_y, target_z"
    "sets the forward orientation such that the node points at the"
    "target point. The target points should be given in global (world)"
    "coordinates"
```
The \textit{set\_orientation} and \textit{set\_forward} methods expect vectors that are oriented with respect to the node’s owner in the scene-graph (or the 3dworld if the 3dnode is filed in 3dworld\textquoteleft node\_set). Likewise, the \textit{set\_location} method provides coordinates with respect to the coordinate system defined by node’s owner in the scene-graph. However, in some cases it may be easier to position the node with respect to its own coordinate system. The \textit{get\_orientation} method will compute the current orientation vector of the object.

The \textit{move} method will shift the position of a node by a movement right, up and forward with respect to its own axes. The node’s location will be moved along its local x-axis, y-axis, and z-axis by the three given values.

\begin{verbatim}
define move as a method given
  3 double arguments      ''dx, dy, dz (right, up, forward)
\end{verbatim}

It is also possible to “spin” a 3dnode on one of its three local axes. The \textit{rotate\_x}, \textit{rotate\_y}, and \textit{rotate\_z} methods will do just that. Each method takes an angle (in degrees) as an argument and spins the 3dnode by that amount about the local (not owner) axis. These methods are similar to the \textit{move} method in that they take “delta” values instead of absolute values. For example, if an airplane is pointed forward along its positive z axis, call the \textit{rotate\_x} method will pitch up or down. In this case the local Y and Z axes are rotated, but the X axis will remain unchanged. Calling the \textit{rotate\_y} method will yaw about its y axis. Calling the \textit{rotate\_z} method will “roll” the airplane.

\begin{verbatim}
define rotate\_x, rotate\_y, rotate\_z as method given
  1 double argument       ''angle in degrees
\end{verbatim}

The local axes of a node are rotated with the node itself. For example, in Figure 67, a box is first rotated about the z-axis the moved by 100.0 units in the “Y” direction (up).
The scale method will modify size of the node. A scaling factor is provided for each axis and, as with move and rotate scaling is performed along the local axes. Each axis is scaled by the given scale factor (a value of “1.0” will not change the axis).

```text
define scale as a method given
  3 double arguments      ''sx, sy, sz (width, height, depth)
```

In cases where a 3dnode must be located in the hierarchy, the find method can be used. This method takes a single text argument and matches it to the name attribute of one of the descendants. First the node_set is searched. If a match is not found, the find method is called recursively for each node in the set. “0” is returned if no match is found.

```text
define name as a text variable
define find as a 3dnode reference method given
  1 text argument
```

Simulation is integrated into the 3dnode by allowing instance to move over simulation time. After the 3dwindow’animate process method is called, all 3dworld instances attached to the 3dwindow’world_set will be automatically and frequently updated as simulation time is scaled to real time (see timescale.v). 3dnode instances can be filed into the 3d.m:motion_set to enable the motion method to be called as time is advanced. The change in simulation time is passed as the argument to motion. Sub-classes of 3dnode can override the motion method making calls to move, set_forward, etc based on the elapsed time.

```text
define motion as a ''virtual'' method given
  1 double argument      ''dt. Time elapsed since last call
```

If simple linear movement is required, the set_velocity method can be called. A velocity vector is given whose components define the speed along the x, y and z axes respectfully. Velocity is set with respect to the owner node’s coordinate system. If a velocity is specified, the set_location method is called automatically by the 3dnode’motion method to update the location. If motion is overridden, 3dnode’motion must be called if the velocity attributes are set. A 3dnode instance must be filed in the motion_set for velocity to take effect.
define velocity_x, velocity_y, velocity_z as double methods
define set_velocity as a method given
  3 double arguments        ''velocity_x, velocity_y, velocity_z

The move_to process method is a convenient way to tell a 3dnode instance to travel to a location in 3d space with a certain speed. The first three arguments specify the location in the owner node’s coordinate system. The fourth argument is the speed, and must be positive. A simulation can “wait for” a node to arrive at the location by calling the move_to method instead of scheduling it. In either case, the instance must be filed in the motion_set before the move_to process method is activated.

define move_to as a process method given
  3 double arguments,    ''x, y, z destination
  1 double argument      ''speed in units/second

define car as a 3dnode reference variable
...
start simulation
file this car in motion_set
call set_location(car)(-100.0, 0.0, 0.0)  ''start at (-100,0,0)
call move_to(car)(100.0, 0.0, 0.0, 200.0)''move to (100,0,0)

A 3dnode object may be used to display a model obtained from a 3dmodel instance. The load method can be called to copy the graphics from an existing 3dmodel instance that has itself been loaded via the 3dmodel_read method. Using the load method instead of assigning the model attribute is necessary if the program needs to modify sub-nodes originally defined by the model. For example, a “tank” model may have a “turret” sub-node which must be rotated with respect to the tank. In this case each tank node in the world may have a different turret orientation or position with respect to the tank, therefore it is not possible to display the same tank image in multiple locations. When load is used, new instances of sub-nodes are copied from the 3dmodel to the node_set owned by the 3dnode. The “find” method can then be used to get a reference to nodes named in the model. The copy_attributes method is automatically invoked for the sub-nodes when the load method is called. If application defined sub-classes of 3dnode are contained in the model’s node_set, copy_attributes should be overridden to ensure that all “new” fields are duplicated.

Another way to define the shape of a node is by assigning its model attribute. During initialization, an instance of a 3dmodel object can be assigned to the model attribute of the 3dnode. The image of the model can then be located and oriented by calling methods such as set_location and set_orientation. Multiple 3dnode objects can reference the same 3dmodel as long as all the objects are attached to the same 3dworld.

define model as a 3dmodel reference variable
  monitored on the left
     ''references the model for drawing this node.

Another attribute of the 3dnode that affects the appearance of objects filed in the node_set is the material.
define material as a 3dmaterial reference variable
'points to 3dmaterial the will be define the default material
'properties for this node and all sub nodes

If sub-nodes have the same appearance or if they all refer to the same texture, assigning this common 3dmaterial to the parent 3dnode will optimize rendering performance. For example, if to draw a section of railroad track, file all railroad ties in a 3dnode and assign the material attribute to the 3dmaterial to be used by each tie.

Options that control how a 3dnode is displayed can be set via the enabled attribute. Although this attribute is defined as a method, it can be used on the left and right, and takes an integer identifier as its argument.

define enabled as an integer method given
  1 integer argument   ''drawing aspect to enable or disable
' enables or disables a drawing attribute. To disable, assign 0. To
' enable assign 1. Pass one of the constants, (_lighting, _visibility).
' method behaves recursively...drawing attributes of all sub-nodes will
''be modified.

The following flags are currently defined:

_visibility – If this flag is zero, the node will disappear the next time the window is updated. Setting this to zero, is useful to temporarily erase a node. DEFAULT: 1
_lighting – If this flag is “1”, lighting calculations will be used to shade this object. When the flag is zero, the node will be fully visible regardless of the position, direction and intensity light sources. Setting this flag to zero is useful for lines and text that are used to label or annotate a node during a simulation. DEFAULT: 0

For example, to erase the node “car” the following code could be used.

Let enabled(3dnode'_visibility) = 0

Another way to erase a node, is to remove it from the node_set. When removed, the node will not be visible when the window’s canvas is refreshed.

3dpoints (3dpoints → 3dgraphic → 3dnode)
The 3dpoints class is derived from 3dgraphic and represents a collection of simple dots in 3d space. The points array attribute is 2-dim array containing the 3d coordinates. The color attribute can be assigned a value returned from the gui.m:color’rgb method and will apply to all points. The size attribute controls the size in pixels of each dot. The default size is “1”.

237
3dtext (3dtext → 3dgraphic → 3dnode)

The 3dtext is derived from 3dgraphic and can be used to show a simple text string within the scene-graph. The string attribute must be assigned to the desired text. The text can be colored and there are several fonts to choose from. Currently, only predefined fonts are available as class attribute pointers that can be assigned to the font attribute. These fonts are automatically initialized before the SIMSCRIPT program is run. The choice of font affects not only the appearance of the text but also its behavior. Both raster and vector fonts can be used:

Vector fonts:
A vector font is rendered by drawing a series of line segments in 3 dimensions. Vector text follows the same rules as do other 3dgraphic shapes with regard to location and orientation. The advantage of using this type of text is that it is “part of” the object, for example the text on a road sign, or the monogram on the side of an airplane. The text will increase in size as the camera moves closer to it. The disadvantage is that the text is usually composed of thin lines and may not look good when it is sized big.

The following vector fonts are available:

- 3dtext’stroke_font - Variable width vector font
- 3dtext’strike_mono_font - Fixed width vector font

The size of vector text is controlled by its width and height attributes. These attributes function the same as the width and height for the 3drectangle class do. The height defines the maximum height including descenders and the width applies to the whole text string.

Characters in a bitmapped or “raster” based font are basically small 2d bitmap images that are copied to the screen when the text is rendered. Text drawn using these types of fonts will always appear right side up regardless of how the 3dtext object (or its owner node) is oriented. However, the 3dnode’location properties is still utilized. In other words, the text can be positioned by calling the set_location method. Bitmapped text will appear the same size regardless of its distance from the camera. If a larger or smaller text size is needed, a different font must be assigned to the font attribute.

- 3dtext’9_by_15_font - 'fixed width bitmap font
For bitmapped fonts, the \textit{align\_horiz, align\_vert} attribute allows the text to be centered, or left/right, top/bottom justified. The following constants can be assigned to the \textit{alignment} attribute:

\begin{verbatim}
define _left_justified=0, _centered, _right_justified as constants ''for the "align\_horiz" attribute
define _bottom=0, _middle, _top, _bottom_cell, _top_cell as constants ''for the "align\_vert" attribute
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{alignment.png}
\caption{3dtext alignment}
\end{figure}

\textbf{3dwindow (3dwindow \rightarrow gui.m:guiitem)}

The 3dwindow represents a window that can be resized or moved much the same as other windows shown on-screen. The window is made up of a resizable frame bordering a \textit{canvas}. It is the canvas that displays one or more \textit{3dworld} instances (which in turn contain the \textit{3dnode} objects composing the scene-graph). Currently, only the 3dwindow class has the capability to display 3d graphics. The 3dwindow can also respond to user events such as window such as an attempt to move, resize or close the window. Asynchronous keyboard input is supported via the 3dwindow.

To display a 3dwindow, an instance should first be created. Since the 3dwindow is the top level item in the scene-graph, it is not filed into any set. The \textit{title} attribute can be assigned to set the title bar text. The window’s initial size and position is controlled by the \textit{position\_xlo, position\_ylo, position\_xhi, and position\_yhi} attributes. Like the \textit{gui.m:window} class, the position is set in \textit{screen} coordinates. The (0,0,0,0) coordinate is located at the bottom left corner of the computer screen while (32767.0, 32767.0) is located in the upper right corner. (The Microsoft Windows “start bar” is ignored). The \textit{color} attribute is also useful for setting the background color. It can be assigned a value obtained from the \textit{gui.m:color} class and by default is set to \textit{gui.m:color’ _black}. The
window is made visible (and updated) by calling its *display* method. The following code displays a 3dwindow in the upper right corner of the computer screen:

Define window as a 3dwindow reference variable

Create window
Let title(window) = "Hello, window"
Let color(window) = gui.m:color'._blue
Let position_xlo(window) = 32767.0 / 2.0
Let position_ylo(window) = 32767.0 / 2.0
Let position_xhi(window) = 32767.0
Let position_yhi(window) = 32767.0
Call display(window)

Instances of the *3dworld* object that are to be shown in the window’s canvas must be filed into the *world_set* owned by the 3dwindow (See figure 70). If more than one 3dworld is being used, all nodes contained in the 3dworld filed last in the *world_set* will be displayed on top of nodes attached to other worlds *regardless of the distance from the camera*. (The depth buffer is cleared before each world is displayed).

![Figure 70: The 3dwindow's relationship to the scene-graph](image)

Sub-classes of 3dwindow can override the *action* method. This method is called automatically in response to a user-driven event. The action method takes a *3devent* reference argument, and the following event ids are currently supported:

<table>
<thead>
<tr>
<th>Event Id</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>_activate</td>
<td>User clicked on window. Window brought to front.</td>
</tr>
<tr>
<td>_close</td>
<td>User clicked on the “X” to close the window.</td>
</tr>
<tr>
<td>_key_down</td>
<td>Pushing down a key on the keyboard</td>
</tr>
<tr>
<td>_key_up</td>
<td>Releasing a key on the keyboard</td>
</tr>
<tr>
<td>_mouse_down</td>
<td>Clicking in the canvas with the mouse.</td>
</tr>
<tr>
<td>_mouse_up</td>
<td>Releasing the mouse button in the canvas.</td>
</tr>
<tr>
<td>_move</td>
<td>Moving the mouse in the canvas.</td>
</tr>
<tr>
<td>_mouse_wheel_forward</td>
<td>Spin mouse wheel away from user (forward).</td>
</tr>
<tr>
<td>_mouse_wheel_backward</td>
<td>Spin mouse wheel toward used (backward).</td>
</tr>
<tr>
<td>_reposition</td>
<td>Dragging the window with the mouse.</td>
</tr>
<tr>
<td>_resize</td>
<td>Resizing the window with the mouse.</td>
</tr>
</tbody>
</table>
The `action` method should return one of the following two predefined constants: `_continue` or `_block`. If `_continue` is returned, the runtime library will handle the event. Returning with `_block` means that the runtime library will take no action in response to the event. For example, to keep the window from disappearing when closed by the user, the overridden `action` method should return with `_block` instead of `_continue`.

In the following example, the `action` method is overridden to respond to the user pressing the “arrow” keys:

```plaintext
Method action(event)
  If id(event) = 3devent'_key_down
    Select case key_code(event)
      Case 3devent'_up_key
        ''handle up key
      Case 3devent'_down_key
        ''handle down key
      Case 3devent'_left_key
        ''handle left key
      Case 3devent'_right_key
        ''handle right key
    Default
      ''handle unknown key
    Endselect
    Always
  End
End
```

Calling the `display` method will make the window visible. If the window is already visible, the canvas will be updated to show a current image for all 3dworlds (and 3dnodes) that are contained in the window. During a simulation there are usually objects changing location, orientation and appearance over time. To make sure that the image of
the scene-graph is kept up-to-date, the animate process method can be activated. The animate method is scheduled to allow the window canvas to be update automatically during a simulation.

```plaintext
define animate as a process method
given 1 double argument    ''run length for animation
```

Animate takes one argument which determines how long (in simulation time units) to run the process method. The method will run indefinitely if “0” is passed. The display method is called repeatedly by animate in between event notices. It is important to know that it at this point the draw method may be called for instances of 3dgraphic objects contained in the scene-graph. Canceling the process method will stop the automatic animation.

3dworld

A 3dWorld acts as a container for lights, cameras, and graphics. One or more 3dworld instances must be created and filed in the world_set owned by 3dwindow. In turn, the 3dworld class owns the following sets:

**camera_set**
The purpose of the 3dcamera class is to show the 3dnode objects contained in the world. Each camera will only “see” objects that are attached to the same world. In order for the view seen by a 3dcamera to be visible, the 3dcamera must be filed in a camera_set. A 3dcamera instance can optionally be filed in a node_set.

**light_set**
3dlight instances must be filed into the light_set before use. A 3dlight instance will only illuminate the 3dworld owning the light_set that it is filed in. A 3dlight instance can also be filed in a node_set if it is to be attached to a visible object.

**material_set**
3dmaterial instances may be used to define the surface characteristics of a 3dfaces or 3dgraphic instance. They must be filed in the material_set owned by the world in which they are used, or alternatively in the 3dmodel ’material_set (the 3dmodel owner must be filed in the same 3dworld’model_set)

**model_set**
A 3dmodel must be filed in the model_set owned by a 3dworld before the 3dnode object(s) that reference it are displayed. Each 3dmodel object instance may be referenced by many 3dnode instances, but both the 3dmodel and the 3dnode instances must belong to sets owned by the same 3dworld.

**node_set**
The node_set contains all objects derived from 3dnode that can be positioned, oriented and viewed. (The hierarchical usage of the 3dworld’node_set is how the scene-graph is defined). Note that 3dcamera and 3dlight instances can optionally
be filed in the node_set (since they are derived from 3dnode). This could be used to implement the view seen from a moving object, a mobile light source, etc.

The ambient_color attribute controls the color and intensity of ambient light throughout the 3dworld. The intensity of the ambient light can be adjusted by using darker colors. (For example, setting ambient_color to gui.m:color’rgb(0.2, 0.2, 0.2) will result in less ambient light).

“Picking” or selection is supported through the 3dworld. The select_node method can be used to locate the (visible) node at a given pixel location in the window’s canvas. The select_node method returns the leaf node under the (x,y) pixel location given in the first 2 arguments. If overlapping nodes are selected, the node closest to the camera is returned. “0” is returned if no node is selected by (x,y).

```plaintext
define select_node as a 3dnode reference method given
    2 integer arguments  ''pixel x, y location
```

The easiest way to use the select_node method is from within the action method of the 3dwindow class. In the following code, the 3dwindow’action method is overridden to receive mouse clicks. The click location is used to “pick” a node by calling select_node for each world in the world_set.

```plaintext
Method my_window’action(event)
    define node as a 3dnode reference variable
    define world as a 3dworld reference variable

    ''respond to the _mouse_down event
    if id(event) = 3devent’_mouse_down
        for each world in world_set
            do
                ''try to pick a node in this world given click location
                let node = select_node(world)(x(event), y(event))
                if node <> 0
                    ''perform action on selection of node
                    call action(node)(event)
                always
            loop

        if node = 0
            ''perform some sort of action on background click
            always
        always
    end
```

If two or more 3dworld objects are filed in the same 3dwindow’world_set, instances filed last will overlap the instances filed first. In other words, all graphics filed in the latter 3dworld will appear on top of all graphics filed in the previous worlds. The 3d coordinates specified by the 3dnode’set_location method (as well as 3d vertices/points) are specified relative to the owner 3dnode of the object. To find the location in “global” coordinates (i.e. the system used to position nodes filed directly in the 3dworld’node_set)
the \textit{get\_location} method can be called. A reference to the \textit{3dnode} instance is the first argument, and the global (x,y,z) location is yielded.

define get\_location as a method given
  1 3dnode reference argument,  
    ''descendant node
yielding
  3 double arguments  
    ''x,y,z in global coordinates
6.3 3dshapes.m Objects

3dbox (3dbox → 3dshape → 3dgraphic → 3dnode)
A 3dBox is used to implement a simple box with width, height and depth. The set_size method can be called to set these parameters. If an instance's material attribute (inherited from 3dnode) is set, it will apply to all sides. However, by using the materials array attribute, each side can have its own distinct color and texture. This array is reserved and destroyed automatically and its elements can be indexed using the following constants defined by 3dBox:

define _front=1, _back, _left, _right, _top, _bottom as constants

The inherited attribute 3dshape 'inside_lighting attribute should be set to “1” if the viewer is intended to be inside the box. (This allowed effects such as a sky and ground to be implemented).

3dcone (3dcone → 3dshape → 3dgraphic → 3dnode)
The 3dcone can be created and filed into a node_set to add a cone shaped object to the scene-graph. The distance from its tip to base can be set by assigning the length attribute. Its radius attribute controls the radius of its base circle. The default value for both dimensions is “1.0”.

3dcylinder (3dcylinder → 3dshape → 3dgraphic → 3dnode)
The 3dcylinder can be created and filed into a node_set to add a cylinder shaped object to the scene-graph. Its size can be adjusted by setting its length and radius attributes. The default value for both dimensions is “1.0”.

3dellipse (3dellipse → 3dshape → 3dgraphic → 3dnode)
The 3dellipse can be created and filed into a node_set to add a circle or ellipse object to the scene-graph. This object can actually be used to draw a variety of shapes depending on the hollow and shape_mode attributes:

<table>
<thead>
<tr>
<th>“hollow” attribute</th>
<th>“shape_mode” attribute</th>
<th>Resulting shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>_full_mode</td>
<td>Solid ellipse or circle. This is the DEFAULT.</td>
</tr>
<tr>
<td>0</td>
<td>_arc_mode</td>
<td>Chord drawn from start_angle to end_angle.</td>
</tr>
<tr>
<td>0</td>
<td>_pie_mode</td>
<td>Solid pie slice Chord drawn from start_angle to end_angle.</td>
</tr>
<tr>
<td>0</td>
<td>_chord_mode</td>
<td>Chord drawn from start_angle to end_angle.</td>
</tr>
<tr>
<td>1</td>
<td>_full_mode</td>
<td>Hollow ellipse or circle.</td>
</tr>
<tr>
<td>Shape Mode</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>_arc_mode</td>
<td>Arc drawn from <code>start_angle</code> to <code>end_angle</code>.</td>
<td></td>
</tr>
<tr>
<td>_pie_mode</td>
<td>Hollow pie slice drawn from <code>start_angle</code> to <code>end_angle</code>.</td>
<td></td>
</tr>
<tr>
<td>_chord_mode</td>
<td>Hollowed chord drawn from <code>start_angle</code> to <code>end_angle</code>.</td>
<td></td>
</tr>
</tbody>
</table>

When the `shape_mode` attribute is not set to `_full_mode`, the `start_angle` and `stop_angle` attributes determine the starting point and range of the pie, arc, or chord. Each angle is measured in degrees counter-clockwise from the positive local x-axis. Defaults values for `start_angle` and `stop_angle` are 0 and 360 respectively.

The `radius_x` and `radius_y` attributes specify the major and minor axis length respectfully. (For a circle, `radius_x = radius_y`, for an elliptical shape `radius_x <> radius_y`).

**3drectangle (3drectangle → 3dshape → 3d.m:3dgraphic → 3d.m:3dnode)**

The 3drectangle is derived from 3dshape and provides an easy way to add rectangular shapes to the scene-graph. Each 3drectangle instance lies in the x-y plane (with the positive z-axis pointing “up” from the front). Since the super-class is 3dnode, a 3drectangle instance inherits the “location” and “orientation” properties allowing it to be rotated and positioned.

The `width` and `height` attributes of the rectangle instance set the size of the object. The `3dshape`’material attribute can be assigned a `3dmaterial` instance to set the color and texture of the front and back faces. If the material has a texture, it may be necessary to cover the rectangle with a sub-rectangle of the full texture image. The `texture_xlo`, `texture_ylo`, `texture_xhi`, and `texture_yhi` can be used to delineate a box within the image that will be mapped to the surface of the 3drectangle. (See Figure 72)
This class is found in the 3dshapes.m module and is the base class for many shape primitive objects that are provided. Basically, a shape is defined for our purposes as an object whose geometry (i.e. points and normal vectors) are computed automatically based on the object’s “size” attributes. Since shapes are derived from the 3dnode class, they can be positioned, oriented, moved over time, etc. Currently available shapes include the 3dbox, 3dcone, 3dcylinder, 3drectangle, and 3dsphere.

The material attribute can be used to specify a 3dmaterial instance to be used on the surface of the shape. (Currently, texture mapping is only supported for the 3dsphere, 3dbox, and 3drectangle shapes). If the texture_name attribute of 3dmaterial is assigned, the texture coordinates will be computed automatically such that the entire texture is mapped to the surface of the shape.

If the inside_lighting attribute is assigned to “1”, all normal vectors will be reversed. This can be useful if the 3dcamera is placed inside the object. By default inside_lighting is zero.

The 3dsphere can be created to show a sphere object in a 3d graphics scene. Its radius attribute controls the size of the sphere. The color of the sphere can be specified by assigning a 3dmaterial instance to its material attribute. If this material’s texture_name attribute is assigned, the texture image will be wrapped around the shape of the sphere. If the 3dcamera will be placed inside the sphere, the inside_lighting attribute should be assigned to ”1”. 

Figure 72: 3drectangle using texture_xlo, texture_ylo, texture_xhi, and texture_yhi
6.4 Miscellaneous Routines

6.4.1 3d.m:handle.3devents.r

define handle.3devents.r as a routine

If a simulation is not running, call this function to do event handling for the 3dwindows. If a mouse or keyboard event has previously occurred, the appropriate “action” method calls will be made.

6.4.2 3d.m:compute_normal_vector

define compute_normal_vector as a routine given
  3 double arguments,   ''x,y,z of 1st (base) point P(0)
  3 double arguments,   ''x,y,z of 2nd point P(1)   (V1 = P(1) - P(0) )
  3 double arguments    ''x,y,z of 3rd point P(2)  (V2 = P(2) - P(0) )
  yielding
  3 double arguments    ''normal vector

Given three points defining a plane, the normal vector to the plane is computed. Assumes counter-clockwise winding (right-hand thumb rule).

6.4.3 3d.m:vector_normalize

define vector_normalize as a routine given
  3 double arguments    ''x,y,z of vector
  yielding
  3 double arguments    ''x,y,z of normalized vector

Normalizes the given vector to make sure that its length is 1.

6.4.4 3d.m:get_screen_size

define get_screen_size as a routine yielding
  2 integer arguments   ''width, height of screen in pixels

Computes and returns the total size of the computer screen in pixels.
7. Deployment Guide
When it is time to deploy or distribute a finished SIMSCRIPT III graphics application,
some extra steps need to be performed to allow the application to run properly on a
computer that does not have SIMSCRIPT III installed on it.
Since the gui.m library is dependent on JAVA, the JAVA runtime environment (JRE)
must be distributed with the application. This can be done by copying the
$SIMHOME/jre folder into the same folder as the executable.
To set up path names correctly, the deployed application should be launched from a
“.bat” file. This file can be double clicked from the windows explorer or launched from a
windows command window. The file should look like this:
------------------------------------------------@echo off
set PATH=.\jre\bin;.\jre\bin\client;%PATH%
set LIB=.;.\jre\lib;%LIB%
my_program.exe
------------------------------------------------------The file “simgraphics.jar” found in $SIMHOME/lib should be copied into the directory
containing the executable. In addition, a folder named “lib” should be created and the
$SIMHOME/lib/images folder should be copied there.
In summary, to deploy a graphics application, do the following:
1) Link the application statically.
2) Make sure all data, “.sg2” and “.jpg” files needed by the application are copied
<application folder>.
3) Copy the entire $SIMHOME/jre folder to <application folder>.
4) Copy $SIMHOME/lib/simgraphics.jar to <application folder>.
5) Copy $SIMHOME/lib/images to the <application folder>/lib folder.
6) Create the “.bat” file needed to launch the application.

249


8. Index

_BMP file, 157
_activate, 25, 165
_always, 154, 223
_block, 167, 242
_bottom, 153
_bottom_cell, 153
_button_dropped, 75, 85
_button_pushed, 75
_cache_model, 132, 231
_centered, 153
_close, 25, 75, 165
_continue, 76, 167, 242
_crop_bottom, 22
_crop_none, 22
_crop_top, 22
_cursor_box, 26
_cursor_icon, 26
_cursor_line, 26
_cursor_none, 26
_data_changed, 75
_directional, 120, 224
_focus_in, 165
_focus_out, 165
_key_down, 25, 165
_key_up, 25, 165
_left_justified, 153
_lighting, 237
_line_loop, 135, 148, 225
_line_strip, 135, 148, 225
_lines, 135, 147, 224
_literal_key, 171
_middle, 153
_mouse_down, 25, 165
_mouse_move, 25, 165
_mouse_up, 25, 165
_mouse_wheel_backward, 165
_mouse_wheel_forward, 165
_never, 154, 223
_once, 154, 223
_points, 135
_polygon, 215
_polygons, 135
_positional, 120, 224
_quad_strip, 135, 215
_quads, 135, 215
_reposition, 25, 165
_resize, 25, 165
_right_justified, 153
_scroll_x, 25
_scroll_y, 25
_smoothing, 132, 231
_spot, 120, 224
_terminate, 76
_top, 153
_top_cell, 153
_triangle_fan, 135, 215
_triangle_strip, 135, 215
_triangles, 135, 215
_visibility, 237
3d graphics, 96
3d graphics classes, 101
3dbox, 134, 155, 246
3dcamera, 111, 212
3dcone, 134, 155, 246
3dcylinder, 134, 155, 246
3dellipse, 134, 155, 246
3detext, 151
3devent, 214
3dfaces, 134, 155, 215
3dgraphic, 96, 134, 220
3dlight, 96, 223
3dlines, 134, 155, 224
3dmaterial, 96, 156, 226
3dmodel, 96, 229
3dnode, 103, 232, 240
3dpoints, 134, 155, 238
3drectangle, 134, 155, 247
3ds file, 126
3dshape, 248
3dshapes.m, 246
3dsphere, 134, 155, 248
3dtex, 134, 155, 238
3dwindow, 240
3dworld, 97, 103, 240, 243
8_by_13_font, 239
9_by_15_font, 239
model_set, 243
modeless dialog, 71
modifiers, 41, 194, 210
motion, 162, 178, 196, 236
motion_set, 176, 236
mouse button input, 168
mouse movement, 167
mouse wheel input, 168
move, 106, 234
move_to, 236
move_to process method, 177
name, 187, 204, 235
no_error, 205
node_set, 103, 232, 243
normal vector, 136
num_interval_x, 202
num_interval_y, 202
num_interval_y2, 202
opaque, 161
option, 202
orientation, 3d, 105
orthographic transformation, 114
palette, 199
palettes, 85
part_color, 203
part_fillstyle, 203
part_font, 203
part_hidden, 203
part_linestyle, 203
part_markstyle, 203
path, 94, 189
pattern, 198, 226
perspective transformation, 113
perspective_angle, 113
perspective_autosize, 113
perspective_ratio, 113
piechart, 54, 199
plot, 54, 60, 200
plot_data, 56, 203
point_size, 204
points, 37, 215
points, 3d, 149
polyline, 148
popup menu, 87
popupmenu, 203
position_xhi, 21, 209, 240
position_xlo, 21, 209, 240
position_yhi, 21, 209, 240
position_ylo, 21, 209, 240
primitives, 35
print, 209
radius, 246
radius_x, 247
radius_y, 247
read, 133, 229, 236
read_sg2, 19, 205
red class method, 185
gbg class method, 185
ggga class method, 185
rotate_x, 106, 162, 234
rotate_y, 106, 162, 234
rotate_z, 106, 162, 234
rotation, 39, 194
scale, 107, 194, 235
scene graph, 103
screen coordinates, 240
scroll, 187
scroll bar thumb, 23
scroll bars, 23
scrollable_h, 23, 209
scrollable_v, 23, 209
select_node, 171, 214, 244
selected, 83, 187
selected_at, 81, 188
selecting 3d objects, 165, 171
set_boundaries, 29, 206
set_color, 142, 221
set_edge_visibility, 223
set_forward, 105, 111, 122, 233
set_location, 111, 233
set_material, 142, 222
set_normal, 219
set_orientation, 105, 234
set_orthographic, 113, 114, 213
set_pen_pattern, 148, 222
set_pen_size, 148, 149, 222
set_perspective, 113, 213
set_point, 219
set_shininess, 222
set_slice, 56, 199
set_speed_course, 49, 196
set_texture_data, 227