GLG Builder and Animation Tutorial

GLG Toolkit
Version 4.5

Generic Logic, Inc.

Generic Logic, Inc.
6 University Drive 206-125
Amherst, MA 01002
USA
Telephone: (413) 253-7491
FAX: (413) 241-6107
email: support@genlogic.com
web: www.genlogic.com

April 8, 2025
Software Release Version 4.5

GLG Builder and Animation Tutorial

Table of Contents

GLG Builder and Animation Tutorial

Preface

This tutorial presents an overview of the tasks involved in creating animated drawings using GLG and using them in a program. It is intended for users that prefer finding things by a trial and error approach rather than reading the manual. Refer to the GLG documentation for further details.

The Tutorial starts with simple animation examples and moves on to examine more complex features of the Toolkit (constraints, etc.). You won't need these advanced features for your first animations, but you may need them later on as you develop more elaborate drawings.

1. Using GLG Builder

Using GLG drawings as widgets

A GLG drawing is a collection of objects created and saved using the **GLG Builder**. This drawing can be loaded into a **C/C++**, **Java, C#/.NET, JavaScript** or **ActiveX** program and animated with the data supplied by a program. A drawing used in a program is referred to as the **Widget** throughout this document.

Any GLG drawing must use a GLG **viewport** object to contain the drawing's graphical objects. A viewport is an encapsulation of the native window object that is used in the Toolkit to contain graphical objects (polygons, circles, etc.) and to provide a drawing surface for them. A viewport may also contain other nested viewports.

To use a drawing as a widget in a program, the drawing must have a viewport named ***$Widget***. This viewport will be displayed in a program when the drawing is used as a widget. A drawing created and saved using the GLG Builder can then be used as a widget in a C/C++, Java, C#/.NET or JavaScript program. It can also be used in a Java bean, .NET User Control and ActiveX Control.

Creating a new widget

When the GLG Builder is first started, it automatically creates a new widget (a viewport named *$Widget*) and moves the editing focus into it.

To get to the top level, use the *Hierarchy Up* button  from the *Control Panel* on the left, or *Traverse, Hierarchy Up* from the main menu. You'll see the outline of the *$Widget* viewport and an informational comment at the top of the drawing (the comment can be deleted from the drawing).

To go back "into" the widget, first select the viewport by clicking on it with the left mouse button. This will show control points in two corners and the center of the viewport, and the *Status Panel* at the bottom will show the viewport's name and object type: ***Name:*** *$Widget,* ***Type:*** *VIEWPORT.*

With the viewport selected, use the *Hierarchy Down* button from the *Control Panel* or *Traverse, Hierarchy Down* from the main menu: this will bring you down into the viewport, so you can start creating objects.

The next section describes menu options for creating a new widget. These options may be used to create a new widget with a specific resize policy and aspect ratio.

Menu Options for Creating a New Widget

The *File, New* menu provides several options for creating a new widget: *Widget (Resize and Stretch), Widget (Resize, No Stretch),* and *Widget (Fixed Scale)*. Each option has a submenu to choose the aspect ratio of the window where the drawing will be displayed at run time, such as *1:1, 4:3, 16:9*. The aspect ratio option allows to select the X/Y ratio of the widget so that the widget may be edited in the Builder using the aspect ratio that matches the window X/Y ratio that will be used at run time. For example, if the drawing is going to be displayed in full screen mode on a 1600x900 pixel display, the *16:9* option should be selected to avoid distorting the widget appearance at run time.

The *Widget (Resize and Stretch)* option creates a resizable widget that stretches the drawing to fit the new widget size when the widget is resized. Stretching the drawing changes the size of objects in the drawing, which may distort the shape of the objects if the widget's X/Y aspect ratio changes. For example, objects with circular shapes in the original drawing may become elliptical.

The *Widget (Resize, No Stretch)* option also creates a resizable widget that changes the size of objects in the drawing when the widget is resized. However, resizing is done in a way that doesn't stretch the objects in the drawing, maintaining the objects' original aspect ratio to preserve objects' shape, so that circles remain circles and do not get stretched. If the widget's X/Y aspect ratio changes, the widget may have padding areas on the sides to preserve the aspect ratio of the drawing displayed inside the widget.

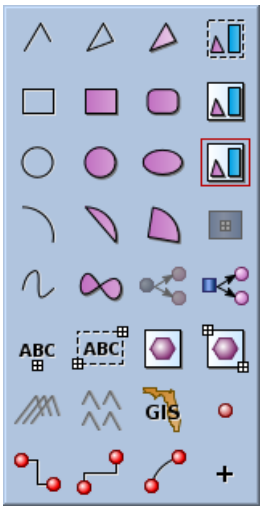
The *Widget (Fixed Scale)* option creates a widget that does not stretch or resize the objects in the drawing, but shows a smaller or bigger part of the drawing area when the widget is resized. This option has a further submenu allowing you to specify the widget width and height in pixels. The object geometry will be defined in pixels and the objects will appear at run time exactly the same as they do in the Builder.

The widget created by default upon the Builder startup uses the option *Widget*, *Resize and Stretch, 1:1 Width/Height Ratio,* i.e. the objects in the drawing will resize and stretch to fit the entire window when the widget is resized.

When a new widget is created using one of the options listed above, the *Resizable*, *Stretch X/Y* and *Span X / Span Y* attributes of the widget viewport's screen are set according to the selected widget creation option.

Creating objects

You can create graphical objects by selecting from a variety of drawing primitives in the *Object Palette* on the left of the Drawing Area:



Object Palette

All buttons in the Builder have tooltips, so you can move the mouse over a button to find its exact function if not sure.

To create an object, select one of the drawing primitives on the left (a *Filled Polygon*  , for example) and click several times in the drawing to define the polygon's points. To finish defining points, click the right mouse button on Windows, the middle mouse button on Linux/Unix, or the *Escape* key on both platforms.

For different objects, a different number of control points may be required. For example, an arc will require 3 control points.

Some object may also require entering additional information. For example, you'll be asked to type a text string when creating a text object.

When objects are being created or any other actions requiring user input are performed, a prompt with additional instructions is displayed in the *Prompt Area* at the bottom.

All objects have to be created inside the widget (*$Widget* viewport).

Editing GLG objects

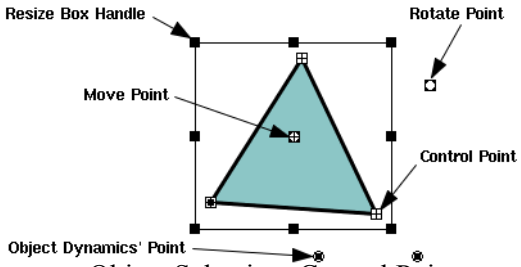
After an object is created, it may be edited in a variety of ways. To edit an object, it has to be selected.

Selecting an Object and Changing Object Geometry

The simplest way to select an object is to click on it with the left mouse button. When an object is selected, its control points and resize box are highlighted in the drawing, and its name and type are displayed in the *Status Panel* at the bottom.

The **move point** appears at the object's center, it's a dynamically calculated point, provided for convenience in the Builder only. You can reposition an object precisely by *Shift*+clicking on the move point and using the arrows in the *Object Move Point* dialog. To avoid accidental movement while you are selecting an object, use *Shift*+click to select the object. For less precise movements, just drag the object with the mouse.

An 8-point **resize box** appears around an object. Use its points to resize the object. You can also flip the object by dragging any of the resize points to the other side of the object's box. Objects with only one control point (marker, fixed text, etc.) can't be resized, and resize points for these objects appear desensitized (in a gray color). *Shift*+click on a resize point to position it precisely with the arrows.



Object Selection: Control Points

A **rotate point** appears on the right side on the resize box. To rotate an object precisely by a specified angle, *Shift*+click on the rotate point and use the arrows in the *Object Rotation Point* dialog. For quicker or less precise rotation, drag the rotate point with the mouse.

Control points appear at the vertices or other important points. To change the shape of the object, drag its control points with the mouse. You can also edit a control point precisely by *Shift*+clicking on it and either using the arrows to move the point, or entering the point's coordinates into the *Value* field of the *Control Point* dialog. Refer to the next section for information on the coordinate system used for the control point coordinates.

If the object has geometrical dynamics attached, the **Dynamics' Points** (such as *Start Point, End Point* or *Rotation Center*) may also be displayed depending on the setting of the *Options, Selection Options, Control Points Display* menu option.

To select an object with no fill (for example, an unfilled polygon), click on the object's edge. The *FillType* attribute controls this aspect of an object's appearance.

Object Geometry and Coordinate Systems

For fixed scale widgets created with the *Widget (Fixed Scale)* menu option, object and control point coordinates are defined in screen pixels. The origin of the coordinate system is located in the upper left corner of the viewport's window, with the Y axis pointing up and the X axis pointing to the right. When the window is resized, more or less of the drawing area is shown, and the use of screen pixel coordinates assures that the objects do not change their size or geometry.

For resizable widgets created with the *Widget (Resize and Stretch)* menu option, object and control point coordinates are defined using the world coordinate system. The world coordinate system has origin in the center of the viewport's window, with the Y axis pointing up and the X axis pointing to the right. The default extent of the world coordinate system is mapped to the extent of the viewport's window and does not change when the window is resized.

The default extent of the world coordinate system depends on the widget's settings:

for widgets with *1:1* aspect ratio, the default extent is [-1000;+1000] in both X and Y directions.
for widgets with *4:3* aspect ratio, the X extent is [-1200;+1200] and the Y extent is [-900;+900]
for widgets with *16:9* ratio, the X and Y extents are [-1600;+1600] and [-900;900], respectively.

The world coordinate system is infinite, and the default extent defines only the mapping of the world coordinate system to the window extent in the unzoomed state, and not the limit on the coordinate values. The viewport's default extent settings are stored in the *SpanX* and *SpanY* properties of the viewport's screen object.

The mapping of the world coordinate system's default extent to the extent of the window is automatically performed by the Toolkit and causes the drawing to be stretched to match the new size of the window when the window is resized. For example, for a widget with 16:9 aspect ratio, the point with world coordinates (1600,900) will always be positioned in the lower right corner of the viewport's window (if the drawing is not zoomed), regardless of the actual window size.

Since the drawing is always stretched to fit the new size of the window, the drawing's aspect ratio may change if the X/Y ratio of the window is changed. As a result, the objects in the drawing may change their shape and circles may be drawn as ellipses.

Widgets created with *Widget (Resize, No Stretch)* menu option also use the world coordinate system but maintain padding areas on the sides of the drawing as needed to preserve the drawing's aspect ratio when the window's X/Y ratio changes. This preserves the shape of objects, making sure that circles remain circles regardless of how the window is resized.

Choosing from Several Selected Objects

When several objects are located close to each other and it is difficult to select the object of your choice, you can use the *Shift* key to help choose among several objects.

Shift-clicking in the drawing area with the left mouse button pops up a menu that allows you to select an object out of several potentially selected objects.

If the *Properties* dialog is open, the arrow button in the upper right corner of the dialog may be used to select an object when several objects are potentially selected.

Using Undo

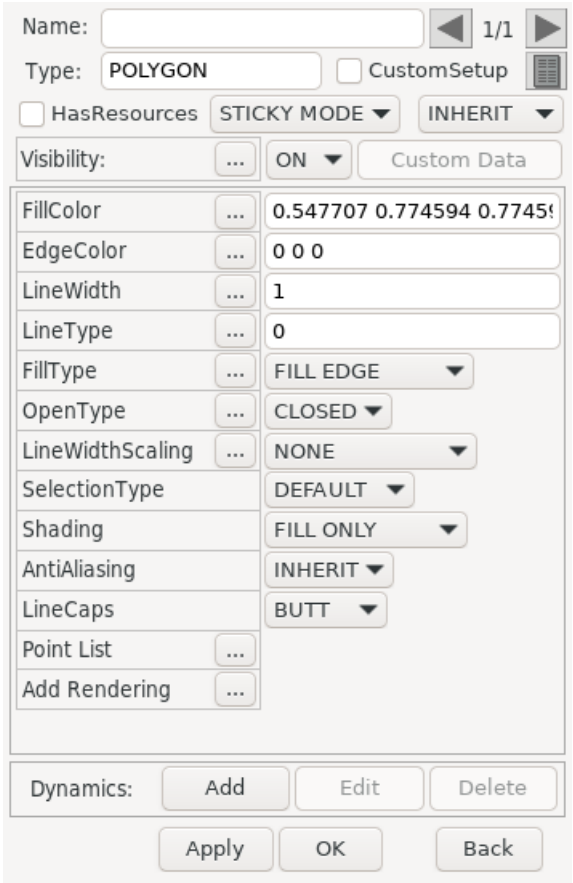
The changes applied to the object's geometry or attributes may be reverted by selecting *Edit*, *Undo* options from the main menu, where the last operation is displayed at the top of the menu. For example, try moving an object with the mouse. To undo the move operation, select *Edit, Undo Object Move or Stretch.*

For multiple step undo, *Edit, Undo History* option may be used. For example, try resizing the object using its resize box and then click and drag one of its control points with the mouse. To reverse one or both of these operations, select *Edit, Undo History, Undo Point Move* to undo the point move operation, and then select *Edit, Undo History, Undo Object Move or Stretch* to undo the resize operation. Notice that the *Undo History* list displays performed operations in the reverse order, so that the last operation appears at the top of the list.

Note: The order of undo steps is important and should be considered when undoing a complex sequence of editing operations.

Editing Object Properties

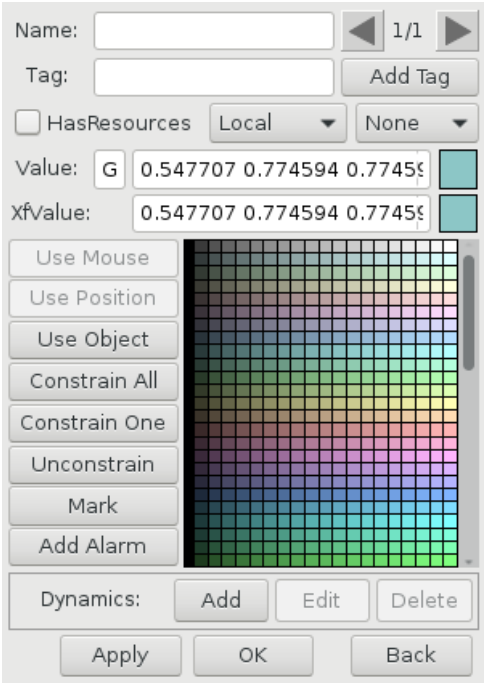
You can use the *Properties* toolbar button to bring up the *Selected Object Properties* dialog. Alternatively, you can use the right mouse button and select *Properties* from a popup menu, or *Object*, *Properties* from the main menu.



Selected Object Properties Dialog

The *Selected Object Properties* dialog allows you to edit an object's properties. Some generic properties common for all objects (*Name, Visibility, HasResources flag*) are displayed at the top of the dialog. Specific properties that depend on the type of the selected object are listed below.

Some properties, for example a *FillColor* property, have an ellipses button  next to them. The button indicates that the attribute is an object, and you can press it to bring a dialog with a color palette and other options for editing the attribute.



Attribute Dialog

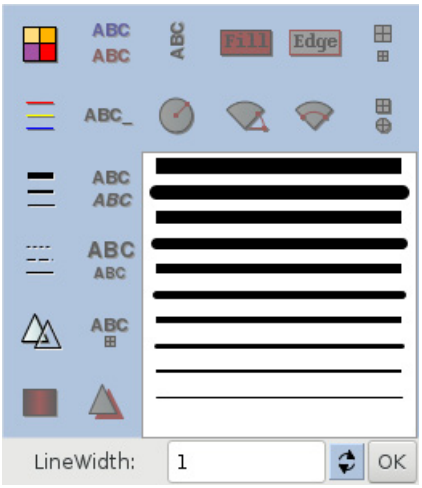
You can select a new fill color from a palette, or give this attribute a specific name that you can later use to access the attribute from a program.

For other attributes, different attribute-related pallets will be shown: try selecting the ellipsis button  for the *LineWidth* and *LineType* attributes.

The *Type* field of the *Attribute* dialog shows the type of the attribute object and may be *D* for double numerical values, *S* for string values or *G* (geometrical) for XYZ or RGB triples used to represent colors and control points in the Toolkit.

Editing objects with the Edit Toolbox

The *Edit Toolbox* can be used as an alternative to the *Properties* dialog for faster editing. Select a polygon in the drawing and click on the *Edit Toolbox* toolbar button to bring the *Edit Toolbox* to edit the polygon.



Edit Toolbox

You can edit various attributes, such as *FillColor* or *LineWidth*. For example, click on the *LineWidth* icon and select a new line width from the line width palette. Notice that the text box at the bottom of the dialog reflects the current setting for the selected attribute.

Only the attribute icons that are applicable to the selected object type are activated in the *Edit Toolbox*, while the other icons are disabled. For example, for the polygon object, the *FillColor* and *EdgeColor* icons are active, while the *TextColor* icons are disabled.

While the *Properties* dialog provides access to all attributes that are available for the selected object, the *Edit Toolbox* displays attributes that are commonly used for a given object type. For example, for a polygon object, you can edit such attributes as *FillColor* or *EdgeColor* via the *Edit Toolbox*, however, the *OpenType* attribute is accessible only via the *Properties* dialog. Likewise, the *GradientType* attribute can be accessed only through the *Properties* dialog, while the value of *GradientColor* attribute can be edited using the *Edit Toolbox*. The *Edit Toolbox* is convenient for a quick and easy editing of the visual appearance of objects, as well as editing multiple objects or all objects in a group. For editing the non-visual entities, such as attribute names, *HasResources* flag and others, the *Properties* dialog should be used.

Multiple Selection

Create three polygon objects using the *Filled Polygon* button. To select multiple objects for editing, click and drag a rectangle to enclose the objects you want to edit. For example, click and drag a rectangle to include all three polygons. This creates a temporary group that includes all selected objects.

Notice that the *Status* panel at the bottom of the Builder's window indicates that the selected object is of type *Group* and its name is *$TempGroup*, which is a predefined name for a temporary group. The resize box of a temporary group is shown as a dashed line. A temporary group is volatile and will be discarded when it is unselected.

To include or exclude an object from the group, you can *Ctrl*+click on the object you want to delete or add. For example, *Ctrl*+click on one of the polygons to delete it from the group. *Ctrl*+click on the same polygon again to add it back to the group.

Let's try to change *FillColor* of all three polygons. Click on the *Edit Toolbox*  toolbar button to bring the *Edit Toolbox* for the selected objects. Click on the *FillColor* button in the *Edit Toolbox* and select a new color in the color palette. All selected objects will reflect the color change.

Press *Escape* to unselect the objects.

In the above example, a temporary group was created using a click and drag operation that defined a rectangular area containing the objects to be selected. This option may be inconvenient in some cases, when there are several intersecting objects and precise object selection is required. Alternatively, a temporary group may be created by *Ctrl*+clicking on each object that needs to be selected.

For example, *Ctrl*+click on one of the polygons with the left mouse button to select it. *Ctrl*+click on another polygon, and then on the third one. All three polygons will be selected. Attribute editing of all selected objects can be done using the *Edit Toolbox*, as described earlier.

Press *Escape* to unselect the objects.

Editing Polygon Attributes

In GLG, the polygon is a fundamental graphical primitive. Most other GLG graphical objects (arcs, connectors, splines, etc.) inherit their attributes from the polygon object. As an example, let's try editing attributes of a filled polygon.

Click on one of the previously created polygon objects to select it, then click on the *Properties*  toolbar button to display the polygon's *Properties* dialog for it.

FillColor and EdgeColor attributes

Click the ellipsis button  for the *FillColor* attribute. A new dialog with caption *FillColor Attribute* comes up, allowing to edit the attribute.

Select a new fill color from the color palette and notice the changes reflected in the graphics. The *Value* field in the attribute dialog shows the new color as RGB values in the range [0; 1].

The *Type* field in the *FillColor Attribute* dialog shows *G,* indicating it is a *G* type (geometrical) attribute, whose value is represented by a triplet of *x,y,z* values. In the case of a color attribute, such as *FillColor* or *EdgeColor*, the triplet represents RGB values.

By default, the color RGB values are displayed in the [0;1] range but it may be changed to the [0;255] range by setting the *ColorDisplay255* variable in the *glg_config* configuration file.

Using similar steps, try changing the polygon edge color using the *EdgeColor* attribute from the *Properties* dialog.

The color may be assigned either from a color palette, or by entering new RGB values directly in the *Value* field of the *Attribute Dialog*, or by typing RGB values directly into the corresponding text box in the *Properties* dialog.

LineWidth and LineType attribute

The *LineWidth* attribute may be entered by typing a pixel line width value into the *LineWidth* text box in the *Properties* dialog. To select a line width from a palette, click on the ellipsis button  for the *LineWidth* attribute and select a line width from a palette.

To select a line type from a palette, click on the ellipsis button  for the *LineType* attribute.

For both *LineWidth* and *LineType*, the *Type* field in the attribute dialog displays *D,* indicating it is a *D* type attribute whose value is a *double* value type.

FillType, OpenType and Shading attributes

The *FillType* attribute controls whether the polygon is drawn as an outline, fill or both. The attribute has an option menu that allows selecting the fill type. The ellipsis button  for the attribute may also be used to pop up the *Attribute* dialog which allows naming the attribute, setting its flags and attaching attribute dynamics.

The *OpenType* attribute controls whether the polygon draws the line connecting the first and last polygon's points, and may be edited in the same way as the *FillType*.

The Builder's *Object Palette* has convenience icons for creating filled, unfilled, open and closed polygons. However, the *FillType* and *OpenType* attributes may be changed after the polygon was created as well.

The *Shading* attribute provides a way to selectively disable 3D shading of the polygon when it is displayed in a viewport with the shading enabled (a *Light* object is added to a viewport). The attribute can also be used to control what parts of the polygon are shaded: just the fill, or both the fill and the edges. This attribute is not an object, so there is no ellipsis button for this attribute. The only available attribute action is changing its value using the option menu.

All of the above attributes are of type *D* (double), as indicated in the *Type* field in each corresponding attribute dialog, and have a numerical value as shown in the *Value* field in the attribute dialog.

PointList attribute

The *PointList* item in the polygon properties is enabled in the *Enterprise Edition* of the Builder and allows adding, deleting and rearranging the order of the polygon points after the polygon has been created. Pressing the ellipsis button ... for the *PointList* displays the list of the polygon's points, with buttons for adding, deleting and rearranging the points' order. Selecting a point in the list with the mouse displays the *Control Point* dialog for editing the point's value, name and flags.

Rendering attributes

The last item *Add Rendering* in the *Properties* dialog allows adding and editing optional rendering attributes, such as gradient fill, cast shadows, arrowheads and fill dynamics. Refer to the *Adding Extended Rendering Attributes* section on page 57 for details.

Defining resources for animation

Overview

Objects in the drawing are animated at run-time using resources, and defining resources for animation is as simple as naming the objects.

A *resource* is simply a name given to an object or an object attribute. For example, naming a polygon object *Poly1* makes *Poly1* visible as a resource providing a handle to the object and allowing to access object attributes.

Each attribute object can be accessed as a resource using a *default attribute name.* For example, *FillColor* attribute of a polygon object can be accessed using a resource name *FillColor*. Likewise, *LineWidth* attribute can be accessed using a resource name *LineWidth*. A complete list of default attribute names may be found in the *Appendix C: GLG Object Attribute Table* chapter on page 394 of the *GLG Programming Reference Manual.*

Once a graphical object is assigned a unique name, its attributes may be accessed using default attribute names using a resource path. For example, fill color of a polygon named *Poly1* may be accessed using a resource path *"Poly1/FillColor"*.

An attribute may be also given a custom name, for example *FillColor* attribute may be named *BackgroundColor*. Details of how resources are used and the purpose of assigning custom names to attributes are described in the sections below.

Using default attribute names for animation

To animate attributes of a graphical object using *default attribute names*, a graphical object itself has to be assigned a unique name.

Let's name a polygon created in the previous section *Poly1*. Select a polygon object, bring its *Properties* dialog and enter *Poly1* into the *Name* field. Notice the *Status* panel at the bottom of the Builder showing *Poly1* in the *Name* field.

Press *ESC* to unselect the polygon, or click anywhere in the drawing's empty space. The *Status* panel at the bottom of the Builder shows ***Name:*** *No name,* ***Type:*** *No object,* indicating no object is currently selected.

Select the *Resources* toolbar button or select *Resources* in the popup menu to display resources of the entire drawing. The *Resource Browser* dialog with the *Drawing Resources* caption comes up, populated with a list of resources for the entire drawing. It includes *Poly1* resource, allowing to access the polygon object using its name, i.e. using a resource *Poly1*. The ">>" symbol in the *Resource Browser* dialog indicates that this is a composite resource containing subresources.

Double-click on the *Poly1 >>* item. The *Resource Browser* now shows a list of subresources for the selected item. Clicking on any resource will bring a resource dialog for it allowing to edit the resource value. For example, click on *FillColor* in the resource list and try changing the color using the color palette. In the *Drawing Resources* dialog, notice the *Selection* field displaying the current resource path *"/Poly1/FillColor"*.

Click on the *".."* item in the list to go one level up, back to the resource list of the whole drawing, i.e. resource list of the *$Widget* viewport. The *Selection* field in the *Drawing Resources* dialog now shows "/", indicating we are at the top level of the resource tree for the drawing.

Once an object has been named, its attributes can be animated at run-time using *default attribute names*. For example, fill color of the polygon *Poly1* can now be accessed at run-time using a resource path *"Poly1/FillColor"*, as shown in the following code sample:

```
GlgSetGResource( viewport, "Poly1/FillColor", r, g, b );
```

where *Poly1* is the object name, and *FillColor* is the default attribute name for the *FillColor* attribute.

Likewise, line width of the polygon *Poly1* can be accessed at run-time using a resource path *"Poly1/LineWidth",* using a default attribute name:

```
GlgSetDResource( viewport, "Poly1/LineWidth", new_value );
```

As shown in the above code samples, resources are strings that represent a *resource path* relative to the specified GLG object. In the above example, the resource path is relative to the *viewport* object, which is the top level drawing viewport.

A concept of a resource path is similar to the concept of a directory or file path on a filesystem: a file path or subdirectory path is relative to the current directory, and '/' character is used as a separator to define a path. Likewise, a *resource path* is *relative* to the specified object and uses '/' as a separator to specify a hierarchical resource path. More information on the resource hierarchy is described below in the *Creating Resource Hierarchy* section on page 25.

Similar to the filesystem, the GLG *Resource Browser* uses the '..' symbol allowing to go one level up in the resource tree.

Prototyping Animation in the Builder

Drawing animation may be prototyped right in the editor using simulated data.

Select *Start*  from the toolbar or *Run, Start* from the main menu to start the *Run Mode*. If the editing focus was inside *$Widget* viewport, starting the *Run Mode* will bring you back to the top level.

A dialog comes up with a sample animation command, which should be modified to specify a resource path to be animated for the current drawing. For example, enter the following animation command to animate line width of the polygon we named earlier *Poly1*:

```
$datagen -sin d 1 5 $Widget/Poly1/LineWidth
```

where:

$*datagen*
: a GLG data generation utility used to animate the drawing with simulated data,

-sin
: selects a sinusoidal waive to generate simulated data,

d
: data type for simulation (*d* for double, *s* for string or *g* for geometrical),

`1 5`
: data range for the generated data values,

$Widget/Poly1/LineWidth
: a *resource name* to animate.

The *$Widget* prefix in the **resource path** is used since the polygon is inside the widget, i.e. the *Poly1* polygon is a child of the *$Widget* viewport. Refer to the *Creating Resource Hierarchy* section on page 25 for details.

Press OK to start animation and observe the polygon changing its line width in the graphics.

Quit *Run Mode* using the *Stop*  toolbar button or the *Run, Stop* menu option.

The animation command may include multiple resources to be animated. For example, to animate the polygon fill color with random RGB values (RGB range is from 0 to 1) in addition to the polygon line width, click *Start*  from the toolbar or *Run, Start* from the main menu, enter the following animation command, and then press OK to start animation:

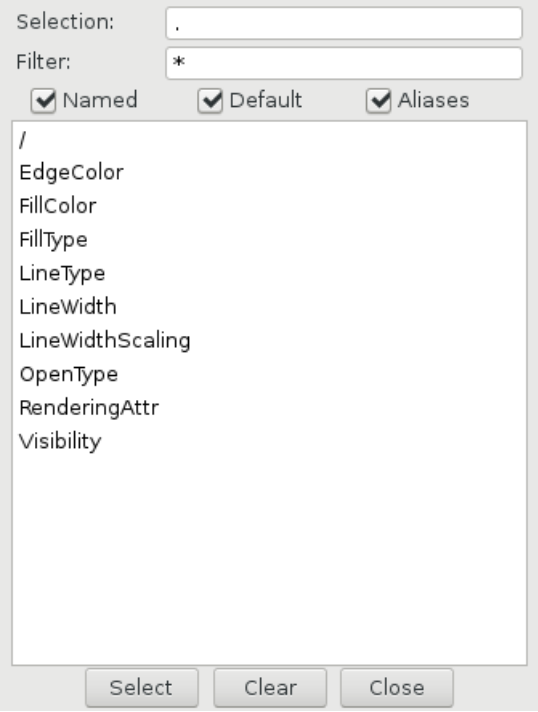
```
$datagen -sin d 1 5 $Widget/Poly1/LineWidth
         g 0 1 $Widget/Poly1/FillColor
```

Select the *Stop*  toolbar button or *Run, Stop* from the main menu to quit prototyping.

For details on the *$datagen* generation utility and the supported parameters, refer to the *The Data Generation Utility* chapter on page 326 of the *GLG Programming Reference Manual.*

Browsing Object Resources

The *Resource Browser* may be used to browse resources of individual objects. Click on the polygon object in the drawing to select it. Bring up the *Resource Browser* dialog for the selected polygon, using either the *Resources* toolbar button, the *Resources* popup menu option, or the *Object, Resources* main menu option. A dialog comes up that lists resources for the selected polygon object, showing *default attribute names* for the polygon attributes, as shown in the following image:



Selected Object Resources Dialog

Notice the "." displayed in the Selection field and the "/" at the top of the resource list, indicating the resource list includes resources relative to the currently selected object, i.e. a polygon.

The *Resource Browser* can be used to edit object attributes. For example, select *FillColor* from the resource list, change the color using the *FillColor Resource* dialog and observe the changes reflected in the graphics.

Close all dialogs.

Assigning names to object attributes

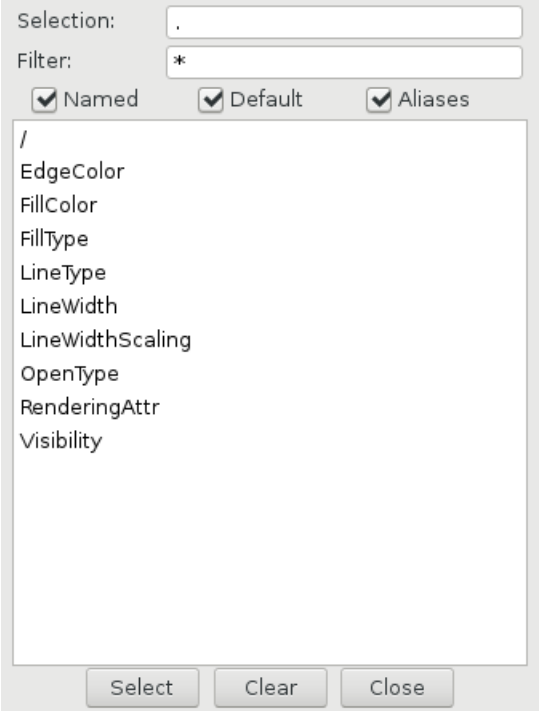
In the previous section, we animated line width and fill color of a named polygon *Poly1* using default attribute names *LineWidth* and *FillColor*.

While each attribute can be accessed using a default attribute name, an attribute can be assigned a custom name. For example, *LineWidth* attribute may be named *BorderWidth*, or *FillColor* attribute may be named *BackgroundColor*.

Select *$Widget* viewport with the mouse and press the *Hierarchy Down* button to go down into it.

Select the *Poly1* polygon and bring its *Properties*. Select the ellipsis button ... next to the *LineWidth* attribute. A dialog with the *LineWidth Attribute* caption comes up, allowing to edit the attribute value or assign a custom name to the attribute. Enter *BorderWidth* in the *Name* field of the *LineWidth Attribute* dialog.This will make *BorderWidth* visible and accessible as a resource.

Bring up the *Resource Browser* dialog for the selected object *Poly1*, using either the *Resources* toolbar button, the *Resources* popup menu option, or the *Object, Resources* menu option. The *Selected Object Resources* dialog will show a list of resources for the selected object, including *LineWidth*, but it will **not** include a resource *BorderWidth*.



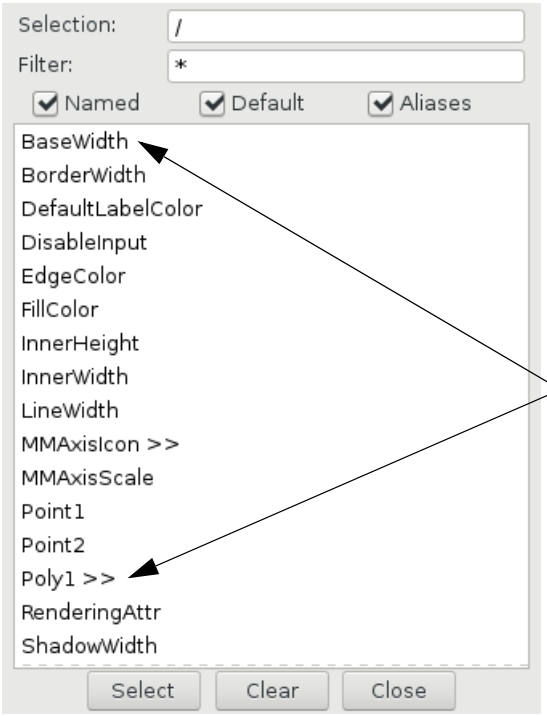
Selected Object Resources Dialog

Close all dialogs.

Unselect the polygon, by clicking anywhere in the drawing's empty space. The *Status* panel at the bottom of the editor should show ***Name:*** *No name,* ***Type:*** *No object.*

Click on the *Resources* toolbar button or use the *Resources* popup menu option to bring up a *Resource Browser* for the entire drawing.

The *Drawing Resources* dialog shows *Poly1* and *BorderWidth* resources.



Entire Drawing Resources Dialog

Notice that these resources appear at the same level of the resource tree, i.e. at the top level of the drawing. Click on the *BorderWidth* resource and try changing its value by selecting a new line width from the line width palette in the *BorderWidth Resource* dialog, and observe the changes in the graphics.

Double-click on *Poly1>>* in the *Drawing Resources* dialog. The browser will show a list of default attribute names for the *Poly1* object, such as *FillColor, EdgeColor, LineWidth.* Click on *LineWidth*. It will bring a resource dialog where the *Name* field shows *BorderWidth*, indicating that the polygon's *LineWidth* has also a custom name *BorderWidth*.

The polygon line width can now be changed in the program using either default attribute name *LineWidth* or an assigned custom name *BorderWidth*. The following function calls will produce the same result:

```
GlgSetDResource( viewport, "Poly1/LineWidth", new_value );
```

or

```
GlgSetDResource( viewport, "BorderWidth", new_value );
```

The first option uses a default attribute name *LineWidth* relative to the *Poly1* resource. The second option uses a custom resource name *BorderWidth* without any reference to *Poly1*. Notice that in both cases, the specified resource path is **relative** to the *viewport* object, a top level viewport of the drawing.

The use of a default attribute name *LineWidth* required a polygon object to be named, while the use of a custom name *BorderWidth* made that resource visible at the top level allowing us to bypass the *Poly1* resource. Even if we didn't name the polygon, we would still be able to set its line width using a resource name *BorderWidth*.

Let's prototype the drawing in the editor using a custom resource name. Select *Start*  from the toolbar or *Run, Start* from the main menu to start the *Run Mode*, enter the following animation command and press OK to start animation:

```
$datagen -sin d 1 5 $Widget/BorderWidth
```

Select *Stop*  from the toolbar or *Run, Stop* from the main menu to quit prototyping.

Following similar steps described above, assign a custom name *BackgroundColor* to the polygon's *FillColor* attribute: select the polygon, bring its *Properties*, click on *FillColor* and enter *BackgroundColor* in the *Name* field. Unselect the polygon, bring up the *Resource Browser* and notice the *BackgroundColor* resource appearing in the resource list, along with *BorderWidth* and *Poly1*. To set the polygon fill color in the program, any of the following options may be used:

```
GlgSetGResource( viewport, "Poly1/FillColor", r, g, b );
```

or

```
GlgSetGResource( viewport, "BackgroundColor", r, g, b );
```

To prototype animation of the polygon's fill color in the editor using the *BackgroundColor* resource, select *Start*  from the toolbar, enter the following animation command and press OK to start animation:

```
$datagen -sin d 1 5 $Widget/BackgroundColor
```

Select *Stop*  to quit prototyping.

Creating Resource Hierarchy

As you recall, both *Poly1*, *BorderWidth* and *BackgroundColor* resources were displayed at the same level of the resource tree in the *Resource Browser*, even though logically resources for the polygon line width and fill color belong to the polygon. This section will explain how to define the resource hierarchy in which the *BorderWidth* resource and *BackgroundColor* belong to the *Poly1* resource.

Make sure you are at the top level of the drawing so that *Hierarchy Up* button is disabled. Unselect *$Widget* by clicking outside of it in the *Drawing Area*, so that the *Status* panel at the bottom displays *No name*, *No object.*

Bring up the *Resource Browser* by using the *Resources*  toolbar button or *Resources* in the popup menu. The *Drawing Resources* dialog includes *$Widget>>* in the resource list.

The *Resource Browser* lists resources **relative** to the currently selected object, and since no object is selected, the *Resource Browser* lists resources of the editor's *Drawing Area*. Since we are at the top level of the drawing hierarchy, with the *$Widget* viewport being inside the *Drawing Area,* its name *$Widget* appears in the *Drawing Area* resource list.

Double-click on *$Widget>>* in the *Drawing Resources* dialog to see resources inside *$Widget*. Resources *Poly1*, *BorderWidth* and *BackgroundColor* are displayed at the same level of the resource hierarchy and are sub-resources of *$Widget*. The resource path for these resources would be *"$Widget/Poly1"*, *"$Widget/BorderWidth"*, *"$Widget/BackgroundColor"*.

Select the widget viewport with the mouse, so that the Status panel displays ***Name:** $Widget, **Type:** VIEWPORT.* Go down into the viewport using the *Hierarchy Down* button.

Select the *Poly1* polygon and bring up its *Properties* dialog.

Set the polygon's ***HasResources*** flag to **YES**.

Bring up *Resources* for the selected polygon. As mentioned above, the *Resource Browser* lists resources **relative** to the selected object, so it now lists resources inside the selected *Poly1* polygon. The list includes *BorderWidth* and *BackgroundColor* as resources inside the polygon (these resources now 'belong' to the polygon), along with the *LineWidth* and *FillColor* resources.

The ***HasResources*** flag defines a **resource hierarchy** for the object. If set to **YES**, all custom attribute names are visible in the resource tree as **subresources** inside the object; in other words, they become **children** of the object.

If ***HasResources*** is set to **NO**, the object is resource-transparent, so that custom attribute names "bleed" through the object and appear as resources at the **same level** as the object in the drawing resource tree.

For example, if *Poly1* polygon has *HasResources=YES, BorderWidth* and *BackgroundColor* become children of *Poly1,* so that the resource path would be *"$Widget/Poly1/BorderColor"* and *"$Widget/Poly1/BackgroundColor"*. However, if *Poly1* polygon has *HasResources=NO, BorderWidth* and *BackgroundColor* will be on the same level as *Poly1,* so that the resource path would be *"$Widget/BorderColor"* and *"$Widget/BackgroundColor",* bypassing a reference to the*Poly1* resource.

The *HasResources* flag controls a resource hierarchy only for the assigned custom attribute names. Default attribute names are always visible as subresources inside the object and do not depend on the *HasResources* flag. For example, a default attribute name *FillColor* may be used only as a subresource relative to the object, i.e. *"$Widget/Poly1/FillColor"* regardless of the *HasResources* flag setting. If we were to access a resource *"$Widget/FillColor"*, it would be a fill color of the viewport itself, not the polygon.

The default value of the *HasResources* flag is ***NO***, meaning that the object is resource-transparent by default.

As mentioned earlier in the tutorial, a resource path is a string representing a hierarchical *resource path* relative to the specified GLG object. Similar to the directory or file path on a filesystem, a *resource path* uses '/' as a separator to specify a hierarchical path.

With the polygon's *HasResources* flag set to **YES**, The following sample code may be used to animate line width and fill color of the *Poly1* polygon in the program:

```
GlgSetDResource( viewport, "Poly1/BorderWidth", new_value );
GlgSetGResource( viewport, "Poly1/BackgroundColor", r, g, b );
```

Create a copy of a polygon by using *Edit, Full Clone* from the main menu or pressing *Control+L* accelerator key to create a copy of the polygon. Name the copy *Poly2*. Move the copy further away from the first polygon.

Bring up the *Resource Browser* and open *Poly1* by double-clicking on it to see its resources. Then double-click on *".."* to return back, and open *Poly2*. You can see that both *Poly1* and *Poly2* have their own *BorderWidth* resources. You can now edit the *BorderWidth* resource of each polygon individually and use *"Poly1/BorderWidth"* or *"Poly2/BorderWidth"* resource names to access the line width of one polygon or another.

Try prototyping using the following animation script to animate the first polygon:

```
$datagen -sin d 1 50 $Widget/Poly1/BorderWidth
```

The following script animates the second polygon:

```
$datagen -sin d 1 50 $Widget/Poly2/BorderWidth
```

However, in the previous section we were also able to animate line width and fill color of the polygon using **default** attribute names with resource paths *"Poly1/LineWidth"* and *"Poly1/FillColor"*, so what would be the advantage of assigning **custom** names *BorderWidth* and *BackgroundColor* to the attributes in addition to having **default** attribute names?

Consider a scenario of creating a custom component composed of a group containing several graphical objects, each having their own resources. In spite of having multiple graphical objects representing a component, it would be desirable to be able to use the component as a single entity with configurable parameters visible at the top level of the component itself. For example, consider having a valve widget, containing a valve symbol and a label to display a valve ID. A valve widget should have resources such as *State* and *ID* visible at the top widget level, so that we could create multiple valves, name them *Valve1, Valve2*, etc. and configure each valve using resources names "*Valve1/State"*, *"Valve1/ID"* and *"Valve2/State"*, *"Valve2/ID"*.

Such components with a flexible resource hierarchy may be created by assigning custom attribute names to necessary attributes and using the *HasResources* flag for the objects inside the component to control where resource names are visible relatively to the component object itself.

More details and examples are described in the following section.

Dynamic Resource Hierarchy

While the object hierarchy is fixed and is completely determined by the parent-child relationship between objects, the *HasResources* flag makes it possible to decouple the resource hierarchy from the object hierarchy, making resource hierarchy dynamic and configurable.

The *HasResources* flag makes it possible to define the place in the resource hierarchy where resources with custom names appear, thus creating custom resource hierarchies with desired resource structure. For example, if several objects are assembled together in a group to represent a custom component, the objects' *HasResources* flags may be used to make resources that control important properties of each object being visible at the level of the component (the group level), instead of being "hidden" inside of the individual objects in the group.

The following steps create a sample component and arrange the objects' *HasResources* flags to create a desired resource hierarchy.

Create a rectangle, bring *Properties* and name it *Box* using the *Name* field. Click on the ellipsis next to the *FillColor* property and name it *BackgroundColor* in the *FillColor Attribute* dialog.

Create a fixed text object by clicking on the *Fixed Text*  button in the *Object Palette*, position it in the middle of the rectangle, enter *Label* in the text string dialog and press *OK* to finish. The *Status* panel at the bottom shows ***Name:*** *No name,* ***Type:*** *TEXT.*

Bring *Properties* for the text object and name the object *LabelObject*. Click on ellipsis next to the *TextString* property and name it *LabelString* in the *String Attribute* dialog.

Group together the rectangle and the text object by clicking on the *Permanent Group* icon  , then click and drag to enclose the rectangle and the text object in the group.The *Status* panel at the bottom shows ***Name:*** *No name,* ***Type:*** *GROUP.*

Bring *Properties* for the group object and name the group *CustomLabel*. Check the *HasResources* toggle to set *HasResources=YES* for the group.

The *Status* panel at the bottom should show ***Name:*** *CustomLabel,* ***Type:*** *GROUP.*

With the group still being selected, bring the *Resource Browser*. It will list the following resources: *Box*, *BackgroundColor, LabelObject, LabelString,* as well as other resources. We can set the background color and label string of the *CustomLabel* component using *"CustomLabel/BackgroundColor"* and *"CustomLabel/LabelString"* resource paths.

Since the *Box* and *LabelObject* have *HasResources=NO,* their named resources *BackgroundColor* and *LabelString* "bleed" through and appear at the top level of the *CustomLabel* component, making them parameters of the component, instead of being parameters of the box and label objects.

If the *Box* and *LabelObject* had *HasResources=YES,* the *BackgroundColor* and *LabelString* resources would appear inside the box and label objects: "*CustomLabel/Box/BackgroundColor"* and *"CustomLabel/LabelObject/LabelString"*. Such long resource paths would be unnecessary and inconvenient for editing the component.

Using Multiple Component Instances

The *CustomLabel* component created in the previous section may be copied to create multiple instances of a custom label. Each copy will have the same resource hierarchy with *BackgroundColor* and *LabelString* resources that are unique for each instance. To differentiate between instances, each copy should have a unique name.

For example, select the CustomLabel component created in the previous section and make several copies using either *Ctrl+L* or the *Edit, Full Clone* menu. Position each copy and name them *CustomLabel1, CustomLabel2, CustomLabel3,* etc.

Press ESC to unselect any selected object. The *Status* panel at the bottom should show ***Name:*** *No name,* ***Type:*** *No object.*

Bring the *Resource Browser* for the whole drawing. It will show all instances as resources: C*ustomLabel1>>, CustomLabel2>>, CustomLabel3>>,* etc.

Double-click on *CustomLabel2>>*. Select *BackgroundColor* with a single-click and change the color using the color palette. Select *LabelString* and change it.

Double-click on *"..."* to go back to the top level, and repeat the previous steps for *CustomLabel3*.

Programming Example

Since the resource hierarchies of all instances are identical, the same code or a function may be used to supply a value (such as a string, color, or a numerical value) to any instance at run time. For example, the following function may be used with the GLG Standard API to set the label string of an instance specified by the *instance_name* parameter:

```
// instance_name may be "CustomLabel1", "CustomLabel2", etc.
void SetLabel( GlgObject drawing, char * instance_name, char * label )
{
   char * res_name =
      GlgConcatResNames( instance_name, "LabelString" );
   GlgSetSResource( drawing, res_name, label );
   GlgFree( res_name );
}
```

If the Intermediate or Extended API is used, the following code may be used to set the label without a need to create and then free a *res_name* string:

```
void SetLabel( GlgObject drawing, char * instance_name, char * label )
{
   GlgObject instance =
      GlgGetResourceObject( drawing, instance_name );
   GlgSetSResource( instance, "LabelString", label );
}
```

Using the Viewport Object

A viewport object is a GLG encapsulation of a window, which may be used to draw other graphical objects. In addition, the viewport object provides its own coordinate system, resizing all objects in the viewport when the viewport is resized. The viewport may be used as a container holding functionally different parts of the drawing, or as a component containing other graphical objects (a graph, control, etc.). Viewports may be used recursively, with one viewport containing a hierarchy of several nested viewports, each containing its own drawing.

To create a viewport, select the *Viewport* icon from the drawing primitives, then click twice in the drawing area to define the position of the viewport's corners. After the viewport was created and selected, use the *Hierarchy Down* button from the *Control Panel* or *Traverse, Hierarchy Down* from the main menu to go "down" into the viewport. You can add any objects to the viewport by creating them while being inside the viewport, then use the *Hierarchy Up* button from the *Control Panel* or *Traverse, Hierarchy Up* from the main menu to go back up to the previous level.

The viewport has its own coordinate system with the origin at the center of the viewport and the Z axis perpendicular to the plane of the viewport's rectangle. The corners of the viewport are [-1000,-1000] and [1000, 1000] in the viewport's coordinate system, and this mapping is maintained when the viewport is resized. The viewport's coordinate system is used to interpret the coordinates of any objects drawn in the viewport. When the viewport is resized, all objects within are resized as well.

Panning and zooming affects the mapping of the viewport's coordinate system. For example, if the viewport is zoomed in to by a factor of 2, the corners of the viewport will correspond to (-500 -500) and (500 500) instead of (-1000 -1000) and (1000 1000) without zooming. The **screen** object associated with each viewport has additional *SpanX* and *SpanY* attributes that control the viewport's coordinate extent and coordinate mapping. Refer to the *Screen* chapter on page 100 of the *GLG User's Guide and Builder Reference Manual* for details.

Setting the editing focus provides a convenient shortcut for quick access to objects inside the viewport without traversing down the hierarchy. To move the focus inside the viewport, click on the *Set Focus* button from the *Control Panel* and click on the viewport, or simply *Ctrl*-*Shift*-click on the viewport. The viewport will be highlighted with thick borders to show it has the editing focus, and you can edit, add or delete objects inside the viewport (**Note:** The Hierarchy Down button is disabled while the focus is inside the viewport). When finished, click on the *Main Focus* button in the *Control Panel*, or *Ctrl*-*Shift*-click outside the viewport to return the focus to the main drawing area.

If a viewport is part of a group, the first *Ctrl*-*Shift*-click on a viewport selects it, and the second *Ctrl*-*Shift*-click moves focus inside the viewport.

Note: *Set Focus* should only be used for quick access to viewport's objects for minor editing. Use *Hierarchy Down* as the primary way to access objects inside the viewport.

The viewport's *FillColor*, *EdgeColor* and *LineWidth* attributes control the background color, border color and border width of the viewport. The viewport's screen *ShadowWidth* attribute controls drawing the shadowed bevels around the viewport's borders. The sign of the *ShadowWidth* controls the type of the bevels: raised shadows for positive values and depressed shadows for negative values.

Integrated Zooming and Panning

In the Builder, the vertical and horizontal scrollbars on the sides of the Drawing Area can be used to scroll the drawing. The drawing can also be scrolled by *Ctrl*+clicking and dragging it with the mouse. To start dragging, the click should happen in an empty area of the drawing. If the whole drawing is completely occupied by objects, the dragging may be started by selecting the *Scroll by Dragging* option of the *View* menu, then clicking anywhere in the drawing and dragging the mouse.

If the drawing needs to be scrolled at run time, the integrated zooming and panning features of a viewport object can be used. Every viewport supports integrated scrollbars for automatic panning that are controlled by the viewport's *Pan* attribute. To enable the scrollbars for a viewport, set its *Pan* attribute to *Pan XY* using the *Properties* dialog. The scrollbars allow the user to scroll the drawing at run time when it extends beyond the boundaries of the viewport's visible area. The scrollbars may be enabled for any viewport object that requires them. If required, only one of the scrollbars can be enabled by selecting the *Pan X* or *Pan Y* setting of the *Pan* attribute.

If the viewport's *ZoomEnabled* attribute is set to *YES*, the viewport object also handles zoom and pan accelerators. For example, *"i"* will zoom into the viewport, *"o"* will zoom out and "*n*" will reset zooming and panning. Scrolling the drawing with the mouse using the *Ctrl-drag* sequence is also supported. Refer to the *Viewport* chapter on page 88 of the *GLG User's Guide and Builder Reference Manual* for a complete list of zoom and pan accelerators. If *ZoomEnabled* is set to *NO*, the accelerator keys are disabled, but the integrated zooming and panning is still available for use programmatically.

To try integrated zooming and panning, draw a few objects inside a viewport with panning enabled, then start the prototyping mode by pressing the *Start*  toolbar button or selecting *Run, Start* from the main menu and entering an empty run command. Click on the viewport with the mouse to move the keyboard focus to the viewport, then press *"i"* several times until the objects displayed in the viewport extend beyond the visible area of the viewport's window. Use scrollbars or the *Ctrl-drag* sequence to pan, then press *"n"* to reset zooming and panning, and press the *Stop*  toolbar button to return to the edit mode.

Refer to the *Integrated Zooming and Panning* chapter on page 238 of the *GLG User's Guide and Builder Reference Manual* for details.

Using GLG Widgets and Palettes

Using Palettes

The *Palettes* menu provides access to palettes of pre-built widgets and objects. You can use these objects by simply dragging them from the palette and dropping them into the drawing. The widgets have built-in dynamics and resources to control their appearance. To simplify widget customization, the majority of the widgets also have built-in public properties that list the most essential resources of each widget.

The *Public Properties* dialog lists the most commonly used widget parameters and is the primary way for editing prebuilt GLG Widgets (except the 2D and 3D Graph widgets, which are edited via resources). The dialog may be accessed via the *Public Properties* toolbar button  , the *Public Properties* popup menu option, or the *Object, Public Properties* menu option.

To reduce cluttering, public properties of widgets with a large number of properties are grouped into categories, such as *Colors*, *ValueLabel*, etc. The *Public Properties* dialog for these widgets will display buttons that provide access to each category's properties. Refer to GLG Widgets Reference Manual for detailed information on widget properties.

The use of public properties makes the widget editing easier and more intuitive for the end user. This is especially important in the GLG HMI Configurator, allowing system integrators to provide OEM ready GLG widgets that can be easily deployed to the end users.

The Resource Browser can also be used to browse and edit widget resources. While the *Public Properties* dialog provides convenient access to the most commonly used widget parameters, the Resource Browser provides access to all widget resources. The resource path displayed in the Resource Browser when browsing widget resources may be used to access the resources via API at run time.

To access widget parameters via resources in a program, each widget should be assigned a unique name, using either the *Properties* dialog or the *Public Properties* dialog. For example, to set the *Value* resources of a dial widget in the program, name the dial *MyDial* and set the value in the program as follows:

Java and JavaScript

```
SetDResource( "MyDial/Value", value );
Update();
```

C#

```
SetDResource( "MyDial/Value", value );
UpdateGlg();
```

C/C++

```
GlgSetDResource( viewport, "MyDial/Value", value );
GlgUpdate( viewport );
```

Alternatively, widgets may be animated in the program using *tags* instead of *resources*. The advantage of using tags is that each widget doesn't have to be named, since tags are global and visible at the top level of the drawing.

Refer to the *Animating Widgets with Data Using Tags* section on page 38 and the *Animating Widgets with Data Using Resources* section on page 35 for information on how to use either tags or resources to animate the drawing with real-time data.

Possible widget palettes that come with the GLG Toolkit include Real-Time Charts, 2D Graphs, 3D Graphs, Controls, Avionics, Process Control Symbols, Electrical and Electronic Circuit Symbols, and Special Widgets palettes. The *GLG Widget Catalog* containing images of all GLG palettes may be found on the GLG web site at *http://www.genlogic.com/widgets.html*.

The *Palettes* menu in the *Community Edition / Demo* version of the GLG Toolkit lists all available palettes of pre-built objects, saved in the demo format. The *GLG Evaluation* version contains only palettes for *Custom Objects* and *Custom Widget Samples.* The *Commercial* version of the *GLG Toolkit* includes palettes of widget sets that were purchased, saved in a production (non-demo) format. If no widget sets were included in the commercial GLG license, only the palettes for *Custom Objects* and *Custom Widget Samples* are installed by default.

For example, to display a palette of push buttons, use a menu option *Palettes, Push Buttons*. To add a button to a drawing, click on a button icon in the palette. A selected button widget will be added to the drawing, positioned in the center of the drawing by default. You can move the widget by dragging it with the mouse, positioning it in the drawing as needed. The widget may be also resized using its resize handles. To assign a unique name to a widget in order to access widget's resources at run-time, bring its *Properties* dialog and enter a unique name in the *Name* field. Alternatively, bring *Public Properties* for the button and enter a unique name in the *Name* field. The widget parameters may be edited using the *Public Properties* dialog. More detailed information, as well as a step by step example, are included below.

By default, clicking on a widget in a palette adds it to the drawing and positions it at the center. The *Palettes, Manual Widget Positioning* menu option can be checked to allow the user to define the widget's position by clicking in the drawing area. The *ManualWidgetPositioning* parameter may be set to 1 in the GLG configuration file (*glg_config* or *glg_hmi_config*) to change the default behavior.

Adding Widgets to a Drawing

To create a new drawing containing several widgets, select *File*, *New, Widget* main menu option, and choose any of the suboptions for now. For example, select *File, New, Widget (Resize and Stretch), 1:1.* For more information on the suboptions, including resizable vs. non-resizable drawings may be found in the *Menu Options for Creating a New Widget* section on page 10.

For example, follow the steps below to create a new dashboard drawing.

Select *File, New, Widget (Resize and Stretch), 1:1.* This will create a new viewport named *$Widget* and will place editing focus inside it.

Select *Palettes, Value Display.*

Click on a *Value Display* widget of your choice in the palette. This will add a selected widget to the drawing as a child of the *$Widget* viewport and position the widget in the center of the drawing by default.

Move the widget with the mouse and position it in the drawing as needed. The widget may be also resized using the resize handles.

The default behavior of the new widget being positioned in the center of the drawing may be overridden using the *Palettes, Manual Widget Positioning* menu option. If the option is checked, you would need to click in the drawing with the mouse to add a new widget at the specified location.

In order to access the widget's resources in a program, each widget should be assigned a unique name. While the widget is selected, click on the *Properties* toolbar button or select *Properties* in the popup menu. This will bring the *Properties* dialog.

Assign a unique widget name by entering *LoadValueDisplay* in the *Name* field.

Add another widget to the drawing, for example a dial. Select *Palettes, Dials and Meters*, and click on any dial of your choice. It will add a dial to the drawing.

Position the dial in the drawing with the mouse.

Click on the *Properties* toolbar button or select *Properties* in the popup menu.

Enter *PressureDial* in the *Name* field to assign a unique name to the dial.

Optional: Repeat the above few steps to add another dial and name it *VoltageDial*.

Adding Widgets to an Existing Drawing

To add a widget to an existing drawing, load the drawing, select its top-level viewport, use the *Hierarchy Down* button to go "down" into it, then use a palette to add widgets.

Editing Widgets

GLG Widgets are highly configurable. Any aspect of the widget appearance or behavior may be configured by editing its parameters, as well as editing geometry of the objects inside the widget.

The widget can be configured using one of the following methods:

- **Public Properties** can be used to configure the most common widget parameters using the *Public Properties* dialog for simplified editing.
- **Resources** provide access to a complete list of all widget parameters via the *Resource Browser*.

While the *Resources* dialog lists all available widget resources, many of the resources are located deep inside the resource hierarchy, which may requires traversing several levels of resource hierarchy to access a particular resource. The *Public Properties* dialog, on the other hand, lists the most commonly used widget properties as a flat list, allowing fast and easy access to widget parameters. Some properties in the dialog are sorted into categories for easier editing. Regardless of whether a widget was edited using resources or public properties, the resulting saved drawing will be the same.

For example, let's edit the dashboard drawing created above using the following steps.

Select the *LoadValueDisplay* widget by clicking on it with the mouse. The current selection is reflected the *Status* panel at the bottom of the Builder that displays ***Name:*** *LoadValueDisplay*.

Click on the *Public Properties* toolbar button or select *Public Properties* in the popup menu.

Try changing the widget's *Description, Units* and *Value* properties and observe the changes reflected in the graphics. If the selected widget has the *AlarmStatus* property, try changing its value between 0, 1 and 2, and observe the changes of the widget colors.

Click on the *Colors* button to access the widget's color properties. Property names starting with *ValueColor* and *BGColor* define the value label and background colors used for each of the alarm states. If present, the *NumStatusColors* property controls the number of alarm states and the number of corresponding alarm colors.

Click on the *DescriptionLabel, UnitLabel* or *ValueLabel* buttons to access label properties of the corresponding labels, such as *FontSize, FontType*, etc.

Select another widget in the drawing, for example, click on the *PressureDial* widget. Notice that the *Status* panel at the bottom of the Builder displays ***Name:*** *PressureDial*, and the *Public Properties* dialog is automatically updated to display parameters of the newly selected object, i.e. *PressureDial*. The *Name* field in the *Public Properties* dialog shows *PressureDial*.

Change parameters of the dial, such as *Low*, *High* or *Units*, and observe the changes in the graphics. For example, set *Low* to *0*, *High* to *100*, *Value* to *70*, *Units* to *PSI.*

Click on the *VoltageDial* to select the object. Notice that both the *Name* field in the *Public Properties* dialog, as well as in the *Status* panel at the bottom of the Builder change to *VoltageDial.*

Using the *Public Properties* dialog, change parameters of the dial. For example, set *Low* to 0, *High* to 15, *Value* to 12, *Units* to *V.*

For comparison, bring the *Resources* dialog for the selected dial, using either the *Resources*  toolbar button or *Resources* in the popup menu. Notice the difference between the list of resources in the *Resources* dialog and the list of public properties in the *Public Properties* dialog. A list of resources is longer and presents a full hierarchical resource list for this widget, while a list of public properties is limited to a subset of resources, presenting most commonly used widget parameters.

Save the drawing using the *File, Save As* menu option. Use *dashboard.g* as the drawing filename.

In the GLG HMI Configurator, which is a simplified editor meant to be used by the end users for creating or editing drawings, the *Properties* button displays the *Public Properties* dialog for objects that have public properties defined. For objects that do not have public properties, the *Properties* button activates the *Properties* dialog that displays default object properties. Most GLG Widgets have public properties defined, and the *Properties* toolbar button (or the *Properties* popup menu option) activates the widget's *Public Propertie*s dialog.

Refer to the GLG Widgets Reference Manual for detailed information on different GLG widgets and their properties.

Animating Widgets with Data Using Resources

Widget resources, such as *Value, Low* or *High,* may be used to set initial values and animate the widget with dynamic data at run-time. For example, the code samples below show how to access resources in the dashboard drawing we created in the previous section, *dashboard.g*.

Code Sample to Animate Widgets Using Resources in the Program

C#/.NET or Java

```
// Initialization.
void Initialize()
{
   // Set Low/High data range for the dials, if different from
   // the values assigned at design time in the editor.
   SetDResource( "PressureDial/Low", 0.0 );
   SetDResource( "PressureDial/High", 100.0 );

   SetDResource( "VoltageDial/Low", 0.0 );
   SetDResource( "VoltageDial/High", 15.0 );
}

// Periodic updates.
void UpdateDrawing()
{
   // Obtain new values for load, pressure and voltage.
   double load = get_load();
   double pressure = get_pressure();
   double voltage = get_voltage();

   // Push data into graphics.
   SetDResource( "LoadValueDisplay/Value", load );
   SetDResource( "PressureDial/Value", pressure );
   SetDResource( "VoltageDial/Value", voltage );

   // Refresh the display.
   Update();     // Use UpdateGlg() for C#
}
```

C/C++

```
// Initialization.
void Initialize()
{
   // Set Low/High data range for the dials, if different from
   // the values assigned at design time in the editor.
   GlgSetDResource( viewport, "PressureDial/Low", 0. );
   GlgSetDResource( viewport, "PressureDial/High", 100. );

   GlgSetDResource( viewport, "VoltageDial/Low", 0. );
   GlgSetDResource( viewport, "VoltageDial/High", 15. );
}

// Periodic updates.
void UpdateDrawing()
{
   double load, pressure, voltage;

   // Obtain new values for load, pressure and voltage.
   load = get_load();
   pressure = get_pressure();
   voltage = get_voltage();
```

```
   // Push data into graphics.
   GlgSetDResource( viewport, "LoadValueDisplay/Value", load );
   GlgSetDResource( viewport, "PressureDial/Value", pressure );
   GlgSetDResource( viewport, "VoltageDial/Value", voltage );

   // Refresh the display.
   GlgUpdate( viewport );
}
```

JavaScript

```
// Initialization.
void Initialize()
{
   // Set Low/High data range for the dials, if different from
   // the values assigned at design time in the editor.
   viewport.SetDResource( "PressureDial/Low", 0.0 );
   viewport.SetDResource( "PressureDial/High", 100.0 );

   viewport.SetDResource( "VoltageDial/Low", 0.0 );
   viewport.SetDResource( "VoltageDial/High", 15.0 );
}

// Periodic updates.
var load,pressure, voltage;
void UpdateDrawing()
{
   // Obtain new values for load, pressure and voltage.
   get_new_values();

   // Push data into graphics.
   viewport.SetDResource( "LoadValueDisplay/Value", load );
   viewport.SetDResource( "PressureDial/Value", pressure );
   viewport.SetDResource( "VoltageDial/Value", voltage );

   // Refresh the display.
   Update();
}
```

Prototyping Widgets in the Builder using Resources

The drawing may be prototyped in the editor by supplying simulated data for widget resources.

Select the *Start*  toolbar button or the *Run, Start* main menu option.

Supply the following animation command:

```
$datagen -sin d 0 100 $Widget/LoadValueDisplay/Value
   -sin d 0 100 $Widget/PressureDial/Value
   -sin d 0 15 $Widget/VoltageDial/Value
```

Click *OK* to start animation.

The widgets will update, showing simulated dynamic data.

Click on the *Stop* toolbar button or select the *Run, Stop* main menu option to quit prototype mode. Note that the editing focus will be at the top level of the drawing. Click on the *$Widget* viewport to select it, then click on the *Hierarchy Down* button to go down into the viewport and continue with the steps described below.

Animating Widgets with Data Using Tags

Any object or widget in the drawing may be animated by pushing dynamic data values to an object attribute, using one of the following methods:

- using the attribute's *resource name*
- using a *tag* attached to the attribute.

While *resources* are hierarchical, tags are global and provide a way to access attributes using a flat structure. The use of resources require assigning a unique name to each widget in order to animate the widget at run-time, as described in the previous section. For example, we had to assign a unique name to a dial widget, such as *VoltageDial*, in order to set the dial value at run-time:

```
SetDResource( "VoltageDial/Value", voltage );
```

However, the use of **tags** instead of **resources** shields an application from the necessity to be aware of the resource hierarchy of the drawing, and doesn't require assigning each widget a unique name. A tag is added to an attribute in a form of a **tag object**. For example, if a tag *Voltage* is assigned to the *Value* resource of the *VoltageDial,* the dial may be animated at run-time as follows:

```
SetDTag( "Voltage", voltage );
```

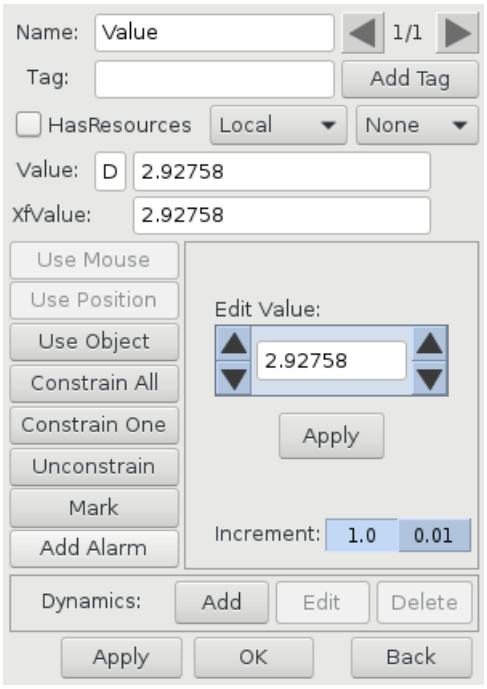
Notice that there is no reference to the dial name, only a tag ID *Voltage* is used to animate the dial.

The steps below describe how to assign tags to the widget resources, using either the *Resources* dialog or *Public Properties* dialog. The end result will be the same.

Assuming the *dashboard.g* drawing is loaded and the editing focus is inside *$Widget* viewport, click on the *VoltageDial* widget to select it.

Bring *Public Properties* for the widget, using the *Public Properties* toolbar button or the *Public Properties* popup menu option. Notice the *Name* field showing *VoltageDial*.

Click on the ellipsis button ... for the *Value* attribute. It will bring the *Value Public Properties* dialog shown below.

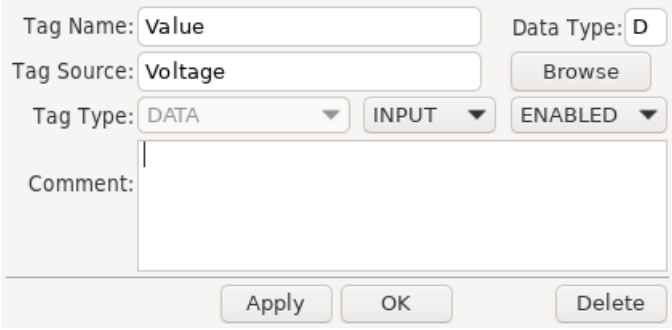


Value Public Property / Resource Dialog

Click on the *Add Tag* button in the *Value Public Properties* dialog. It will bring the *Data Tag* dialog, representing a *tag object* added to the *Value* attribute. If the widget's *Value* attribute already has a tag, the button will be labelled *Edit Tag*. Click on the button to edit the tag.

Tag object has several attributes of its own, including *TagSource*, *TagName*, *TagComment*. *TagSource* is the most essential attribute and points to a *data source variable*, or a *Tag ID*, in the application's data acquisition system. *TagSource* provides a **data connectivity link** between the graphics and real-time data. More details on the tag object attributes may be found in on the *Tag Browser, Tag Object, TagName, TagSource and TagComment* section on page 41.

In the *Tag Source* field, enter a data source name, or a *Tag ID*, known to your application. For example, enter *Voltage,* as shown below:



Data Tag Dialog

In the *Public Properties* dialog, notice the *T* button next to the *Value* property. Clicking on the *T* button will open the *Data Tag* dialog for this property, allowing to edit or delete the tag.

Close all dialogs.

Let's add a tag to the *Value* attribute of the *PressureDial* and use the *Resources* dialog this time. Click on the *PressureDial* widget to select it.

Click on the *Resources* toolbar button or select *Resources* in the popup menu.

Select the *Value* resource. It will bring the *Value Resource* dialog.

Click on the *Add Tag* button in the *Value Resource* dialog. It will bring the *Data Tag* dialog. If the widget's *Value* resource already has a tag, the button will be labelled *Edit Tag*. Click on the button to edit the tag.

In the *Tag Source* field, enter a data source name, or a *Tag ID*, known to your database for the pressure value. For example, enter *Pressure.*

Close the *Data Tag* dialog. Notice that in the *Value Resource* dialog, the *Edit Tag* button is enabled, allowing to edit the tag.

Let's add a tag to the *Value* attribute of the *LoadValueDisplay* widget. Click on the *LoadValueDisplay* widget to select it. Add a tag *Load* to the *Value* resource, using either the *Resources* dialog or *Public Properties* dialog. For example, bring *Public Properties*, click on the ellipsis button  for the *Value* attribute, click on *Add Tag*, and enter *Load* in the *Tag Source* field.

Close all dialogs.

Save changes in the drawing *dashboard.g,* using the *Save*  toolbar button or *File, Save* main menu option.

Code Sample to Animate Widgets Using Tags in the Program

C#/.NET or Java

```
// Periodic updates.
void UpdateDrawing()
{
   // Obtain database values for load, pressure and voltage.
   double load = get_load();
   double pressure = get_pressure();
   double voltage = get_voltage();

   // Push data into graphics.
   SetDTag( "Load", load );
   SetDTag( "Pressure", pressure );
   SetDTag( "Voltage", voltage );

   // Refresh the display.
   Update();
}
```

C/C++

```
// Periodic updates.
void UpdateDrawing()
{
   double load, pressure, voltage;

   // Obtain database values for load, pressure and voltage.
   load = get_load();
   pressure = get_pressure();
   voltage = get_voltage();

   // Push data into graphics.
   GlgSetDTag( viewport, "Load", load );
   GlgSetDTag( viewport, "Pressure", pressure );
   GlgSetDTag( viewport, "Voltage", voltage );

   // Refresh the display.
   GlgUpdate( viewport );
}
```

Prototyping Widgets in the Builder using Tags

The drawing may be prototyped in the editor by supplying simulated data using tags instead of resources.

Select *Start*  toolbar button, or *Run, Start* main menu option.

Supply the following animation command:

```
$datagen -tag -sin d 0 100 Load
         -tag -sin d 0 100 Pressure
         -tag -sin d 0 15 Voltage
```

Click *OK* to start animation.

The widgets will update, showing simulated dynamic data pushed to the specified tags.

Click on the *Stop*  toolbar button, or select the *Run, Stop* main menu option, to quit prototype mode. Note that the editing focus will be at the *$Widget* viewport, at the top level of the drawing. Click on the *Hierarchy Down*  button to set editing focus inside the viewport and continue with the next steps described below.

Tag Browser, Tag Object, TagName, TagSource and TagComment

The tags for each object, as well as for the entire drawing, may be viewed and edited using the *Tag Browser* dialog.

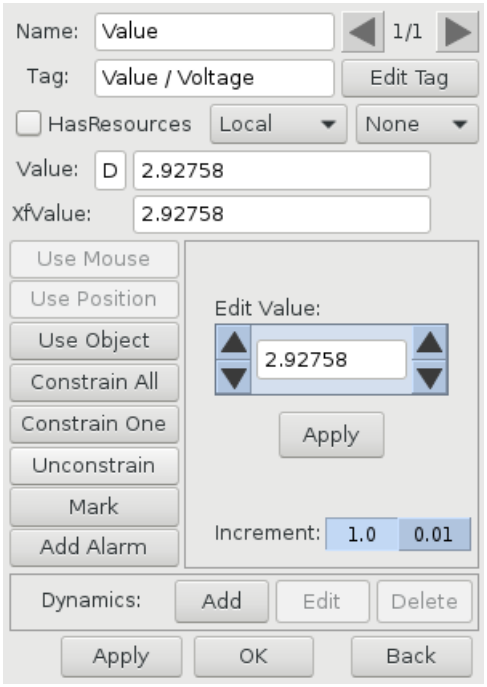
Assuming the *dashboard.g* drawing is loaded and the editing focus is inside the *$Widget* viewport, click in an empty space in the drawing or press *ESC* to make sure no object is selected. The *Name* field in the *Status* panel at the bottom of the editor should show *No name.*

Click on the *Tags* toolbar button  or select *Tags* in the popup menu. This will bring the *Tag Browser* dialog that lists all tags stored in the drawing as shown in the image below.

Tag Browser Dialog

Click on the first entry *Value / Voltage.* It will bring two dialogs:

1. A dialog with the *Attribute with Tag* caption that represents an **attribute (resource) object** the *Value / Voltage* tag is added to:

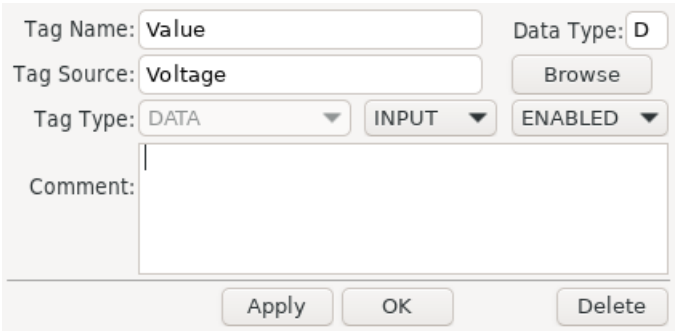


Attribute with Tag Dialog

Notice the *Tag* field showing *Value / Voltage*. Clicking on the *Edit Tag* button will bring the *Data Tag* dialog discussed below, if it hasn't been up already.

2. A dialog with the *Data Tag* caption that represents a GLG **tag object** corresponding to the selected *Value / Voltage* tag.

As mentioned in the previous section, a tag is added to an attribute in a form of a **tag object** and is represented by the *Data Tag* dialog shown in the image below.



Data Tag Dialog

As shown in the *Data Tag* dialog, a **tag object** has several attributes of its own, including *TagName*, *TagSource* and *TagComment*.

TagSource is a string attribute that has a dual purpose, providing a **data connectivity link** between the graphics and real-time data:

- it defines a **data source variable** *(*or a **Tag ID**) in the application's data acquisition system used to supply real-time values for the attribute the tag is attached to.
- it servers as an ID a program uses to push data (obtained from the data source variable defined by *TagSource*) into graphics.

For example, in the *dashboard.g* drawing, *Voltage* is a tag ID the program obtains data from. It is also the *TagSource* the program uses to push the obtained data into the *Value* resource of the *VoltageDial* via the *SetDTag* method:

```
double voltage = GetData( "Voltage" );
SetDTag( "Voltage", voltage );
```

The same tag source string may be assigned to multiple attributes in the drawing. A single API call *SetDTag(tag_source, value)* for a given datasource variable *tag_source* will update **all attributes** with *TagSource = tag_source.*

TagName is a string attribute that may be used to assign a meaningful label to a tag. When a tag object is added to an attribute, the *TagName* is assigned to the attribute's resource name by default. For example, a tag object added to the *Value* attribute has *TagName = Value* by default. *TagName* may be assigned a unique string which can be used by the application to identify the tag object, allowing the application to reassign a data source for this tag object at run-time. While *TagSource* may change at run-time, *TagName* is persistent and does not change when the tag's data source is modified.

In the *dahsboard.g* drawing, all tags have default *TagName = Value.*

The *TagComment* attribute is a string that may be used to store any additional user-defined information associated with the tag.

Custom Tag Data Browser

Often times, defining tag sources manually is not convenient, and a user wants to be able to browse available variable names from a custom data source and select an appropriate variable name for a tag source. For that purpose, the *Browse* button is provided next to the *TagSource* field of the *Data Tag* dialog. Clicking on a *Browse* button brings a *Data Browser* dialog populated with a list of tags defined by the custom data DLL, *glg_custom_data.dll* on Windows, and by a shared library *libglg_custom_data.so* on Linux/Unix platforms.

Custom data DLL should generate a list of available tags as defined by the custom data acquisition system. The returned list of tag names is used by the Builder to populate the *Data Browser* dialog allowing the graphics designer to choose an appropriate tag source for a given dynamic object attribute, such as *Value*.

A sample of a custom data browser DLL, including the source code, is provided with the GLG release and may be found in the directory *<glg_dir>/editor_extensions/data_browser_example.* The example may be used as a template for writing a custom data browser DLL. Refer to the *README.txt* file in this directory for detailed instructions.

Copy *<glg_dir>/editor_extensions/data_browser_example/glg_custom_data.dll* (or *libglg_custom_data.so* on Linux) to the *<glg_dir>/bin* directory.

In the *Data Tag* dialog, click on the *Browse* button next to *TagSource* field. A data browser dialog comes up. Select one of the fields from a list of available tag sources, for example *controller1/group11/tag111*. This tag source gets assigned to the tag. Change the tag source back to the original string, to continue with the rest of this tutorial.

Mapping Tags in an Application at Run-Time

An application can also map tags to datasources at run-time using the GLG API. An application can use the *GlgGetTagList* method to obtain a list of tags defined in the drawing, traverse the list and remap each tag in the list by changing its *TagSource* attribute. The *TagName* attribute of each tag may be used by an application to identify the tag.

Loading Widget Drawings into the Builder

Each widget is represented by a separate *.g* drawing file. Some widgets, such as dials, meters and charts, are contained in its own *$Widget* viewport and may be used in a program "as is", without adding them to a larger drawing. Note that there are some "viewportless" widgets that are not contained in a viewport.

A viewport-based widget may be used "as is", or it can be customized and saved as a custom widget. To customize a widget, bring a desired widget palette and use *Control*-click to select the widget. This will discard the current drawing and will load the drawing (*.g* file) of the selected widget. Customize the widget by editing its resources or public properties, then save the drawing as a custom widget. The filename of this drawing can be used in a program to load the widget at run time.

In addition to the widget itself, the widget's drawing also contains an icon and an animation command for prototyping the widget in the Builder's Run mode as described below.

Alternatively, to create a drawing with only one widget in it, select *File*, *New* (instead of *File*, *New, Widget*), place a single widget in the drawing by clicking on it in a palette, name it *$Widget* and save the drawing.

Refer to the *Adding Widgets to a Drawing* section on page 33 for information on how to add a widget to an existing drawing.

Prototyping Widgets with Predefined Animation Commands

To prototype a widget's run-time behavior, display a widget palette and *Control*-click on it to load the widget's drawing. Then click on the *Start* toolbar button and press *OK* to accept the default animation command. Use the *Stop* button to return to the editing mode.

To test interactive behavior of button, slider or dial widgets, click on the *Start* toolbar button and press *Skip Command* to start prototyping without animation. Click on the widget to test its interactive behavior. For dial and slider widgets, click and drag to move the slider or dial with the mouse. Use the *Stop* button to return to the editing mode.

Disabling Widget's Interactive Behavior

Widgets such as dials and meters have an interaction handler attached to the widget's viewport to handle user interaction, allowing the user to change the widget's value by moving the dial's needle with the mouse. The widgets may be used for both input and output. To disable the widget's interactive capabilities and use it only for output, delete the widget viewport's input handler.

To delete the widget's interaction handler, select the widget in the drawing, bring its *Properties* dialog and delete the handler by emptying the *Handler* text field.

Saving the Drawing

To save the drawing, use either the *Save* toolbar button or *File, Save* from the main menu.

The *Options* menu contains settings that control how the drawing is saved. The default *ASCII* save format enables the drawing to be displayed on different hardware platforms, as well as in the Java, C#/.NET and JavaScript versions of the Toolkit. The default *Save Compressed* option minimizes the size of the drawing. Disable the drawing compression if the drawing is used for code generation.

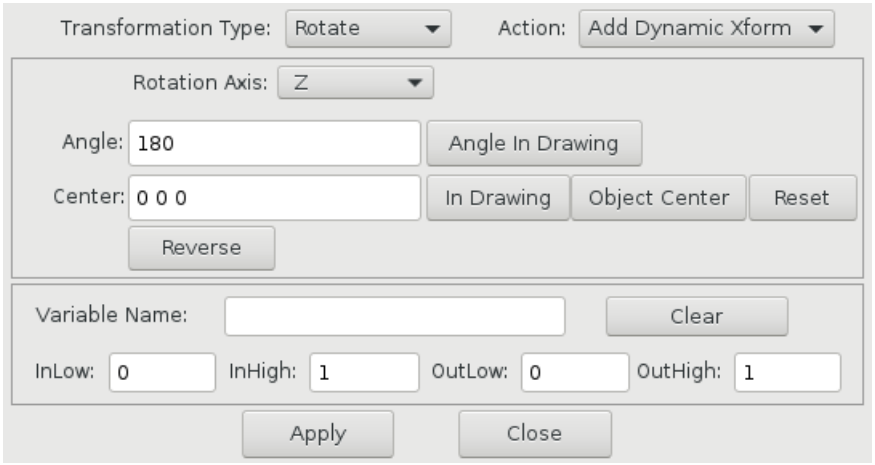
Notice that *LWValue* is a global name. The application doesn't need to know which object or attribute it is connected to.

Adding Geometrical Dynamics

You can add 2D and 3D dynamic behavior by adding dynamics to objects. The available geometrical dynamics include *rotate, scale, move, path* and other types of dynamics.

Let's add rotation dynamics to the *Poly1* polygon. Select the *Poly1* polygon and use *Add Dynamics* from the toolbar, or *Object Dynamics, Add Dynamics* from either the popup menu or the *Object* menu, to bring up the *Add Dynamics* dialog.

Select the *Rotate* transformation type at the top of the *Add Dynamics* dialog, and select the *Z* rotation axis to rotate in 2D:

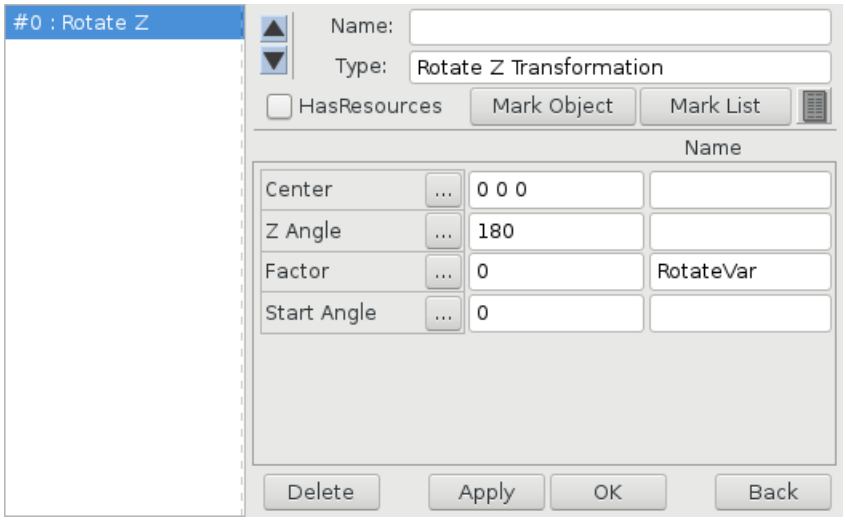


Add Dynamics Dialog

You can either enter the rotation angle in degrees, or define it by selecting *Angle In Drawing* and selecting two points anywhere in the Drawing Area. Try entering 180 as the angle value.

You can also change the center of rotation from its default position by either entering its new coordinates (in the range -1000 to 1000) or selecting the *Center In Drawing* button and clicking in the drawing with the mouse to define a new position.

Type *RotateVar* as the *Variable Name* and click on *Apply*. This will attach the dynamics to the object and will bring up the *Edit Dynamics* dialog that shows dynamics attached to the object. The dialog also shows the resource names assigned to the dynamics' parameters that will later be used to animate or change them: notice that *RotateVar* is used as a resource name of the dynamics' controlling *Factor*.



Edit Dynamics Dialog

The *Factor* parameter is a normalized value in the range [0; 1], and *ZAngle* defines a rotation angle in degrees. Either the *ZAngle* or *Factor* parameter may be changed, and the actual rotation angle is calculated as multiplication of the *ZAngle* and *Factor* parameters.

For example, if *ZAngle*=180 and *Factor*=1, the object is rotated by 180 degrees; setting *Factor*=0.5 rotates the object by 90 degrees.

The *Start Angle* parameter defines the starting position for object rotation. For example, if *StartAngle*=45, the object will be rotated by 45 degrees from its original position first, and then an additional rotation will be applied according to the values of the *Factor* and *ZAngle*.

Prototype the dynamics (*Start* icon from the toolbar) with the following animation script:

```
$datagen -sin d 0 1 $Widget/Poly1/RotateVar
```

It will change the *RotateVar* parameter in the range [0; 1], causing the object to rotate in the range [0; 180] degrees (as defined by the Z*Angle* parameter). Also notice the use of the resource hierarchy since *Poly1* has the *HasResources* set to YES.

Stop the animation (*Stop* toolbar button), go down into the widget viewport and select the *Poly1* object. Activate the *Edit Dynamics* dialog by selecting *Edit Dynamics* from the toolbar (you can also select Dynamics *Edit* from the bottom of the *Properties* dialog, or use *Object Dynamics, Edit Dynamics* from either the *Object* menu or the popup menu).

The *Edit Dynamics* dialog shows the number and types of dynamics attached to an object (any number of dynamics may be attached), as well as parameters of the selected dynamic transformation. You can edit the dynamics' parameters after it has been created by either entering numerical values or selecting the ellipsis button next to the parameter for more options.

For the rotate transformation, either the *ZAngle* or *Factor* parameter may be used as a controlling parameter for animation:

1. Set *ZAngle* to a fixed angle and animate the controlling *Factor* in the range [0;1] to rotate the object by the fraction of the angle. For example, if *ZAngle*=90, setting *Factor*=1 rotates the object by 90 degrees, and *Factor*=0.5 rotates the object by 45 degrees.

   Refer to the *Changing attribute range* section on page 55 for information on how to define a custom range for the *Factor* parameter.

2. Set *Factor*=1 and animate the *ZAngle* parameter by setting it to the actual desired angle of rotation in degrees.

While animating the *ZAngle* is specific to the *Rotate* dynamics, animating the controlling *Factor* may be used as a universal way to animate geometrical dynamics of any type.

If the object is moved, the rotation center moves with the object. For example, if the rotation *Center* coincides with one of the triangle's corners, the triangle will still rotate around that corner after the triangle has been moved to a new position.

This behavior is controlled by the default STICKY MODE setting of the *MoveMode* attribute. If it's not a desired behavior and the object's rotation center should not move with the object, the STICKY MODE setting can be changed to MOVE POINTS at the top of the object's *Properties* dialog. When *MoveMode*=MOVE POINTS, the object will still rotate around the old center position after the object has been moved.

Adding Attribute Dynamics

In addition to setting attribute values directly as it was done in earlier examples, you can also add attribute dynamics to the attribute object in a way similar to adding geometrical dynamics to objects. The GLG Builder provides two sets of attribute dynamics options: predefined dynamics as well as stock transformations. The stock transformations are basic transformation types used as building blocks to implement certain dynamic behavior. Predefined dynamics provide easy to use options for implementing the most common dynamic actions; they represent a collection of several stock transformations wired together to implement a specific dynamic behavior.

The types of available dynamics differ for different attributes. The following sections describe some of the predefined dynamics as well as stock transformations. Refer to the GLG documentation for the complete list of available attribute dynamics.

Predefined Attribute Dynamics

Predefined dynamics provide a simplified interface for adding most common dynamic behavior to object attributes. For example, follow the steps below to add *Color Blinking* dynamics to the *FillColor* attribute of the polygon *Poly1*.

Go down into the widget's viewport, select the *Poly1* polygon and open its *Properties* dialog. Select the ellipsis button  next to the *FillColor* attribute to activate the *Attribute* dialog.

Select the *Add Dynamics* button at the bottom of the *Attribute* dialog. This will display a list of available dynamics, with predefined dynamics listed first by default. The *Stock Dynamics* button at the end of the list may be used to access additional stock transformations described later in this section.

Select *Color Blinking* from the list. *Edit Color Dynamics* dialog appears, presenting a list of parameters of the dynamics. When the *Enabled* parameter is set to 1, blinking is activated at run-time, alternating the object's color between *OffColor* and *OnColor*. When *Enabled* is set to 0, blinking will be disabled, and *OffColor* will be used to display the object's color. The *Interval* parameter defines blinking interval in seconds. For example, set the *Interval* parameter to 0.25 to define a blinking rate of four times per second.

By default, *OffColor* is set to the value of the *FillColor* attribute, and *OnColor* is set to "1 0 0" (red color). These parameters may be changed as needed.

In order to access a dynamics' parameter in the application code, the parameter has to be named. Assigning a name to a parameter makes it accessible as a resource. A name may be assigned to a parameter using the right most field in the parameter's row.

In the *Edit Color Dynamics* dialog, enter *BlinkEnabled* in the right most field in the *Enabled* attribute row. This will allow the application to enable or disable blinking at run-time by setting a resource "*Poly1/BlinkEnabled"* using the GLG API.

Likewise, you can assign names to other attributes. For example, enter *BlinkOffColor* in the right most field in the *OffColor* attribute row. Enter *BlinkOnColor* in the right most field in the *OnColor* attribute row.

Click on the ellipsis button next to the *OnColor* attribute to activate *OnColor Properties* dialog. Notice that *BlinkOnColor* appears in the *Name* field of the *OnColor Properties* dialog, displaying a name assigned to the *OnColor* attribute of the dynamics.

In the *OnColor Properties* dialog, select a color from a color palette to change the value of the *OnColor* attribute.

Note that since the *FillColor* attribute has dynamics attached, the object's color may be now changed only via the *OffColor* attribute of the dynamics. Click on the ellipsis button next to the *OffColor* attribute to activate *OffColor Properties* dialog. Notice that *BlinkOffColor* appears in the *Name* field of the *OffColor Properties* dialog, displaying a name assigned to the *OffColor* attribute of the dynamics.

Select a color from a color palette to change the value of the *OffColor* attribute and notice that the polygon's fill color will change accordingly.

Click on the *OK* button in the *Edit Color Dynamics* dialog to close the dialog.

Click on the *Resources* toolbar button to open the *Resource Browser* for the *Poly1* object. *BlinkEnabled*, *BlinkOffColor* and *BlinkOnColor* are visible as object's resources and may be accessed in the application code using the GLG API.

Click on the *Close* button in the *Resource Browser* dialog to close the dialog.

Click on the *Start* toolbar icon to prototype blinking dynamics. Click on the *OK* button to accept the existing animation command.

Since *Poly1/BlinkingEnabled* is set to 1, the polygon will blink alternating between two colors defined by "*Poly1/BlinkOffColor"* and "*Poly1/BlinkOnColor"* resources.

Stop the animation (*Stop* toolbar button).

Go down into the widget's viewport, select the *Poly1* polygon and bring up the *Properties* dialog.

Notice that there is an *x* button to the right of the *FillColor* attribute, indicating that this attribute has dynamics attached. This button provides a shortcut to the *Edit Dynamics* dialog, allowing to edit or delete the dynamics.

Click on the *x* button next to the *FillColor* attribute to open the *Edit Color Dynamics* dialog.

Click on the *Delete* button in the *Edit Color Dynamics* dialog to delete the *Color Blinking* dynamics.

Many other types of predefined dynamics are available. The following lists some of the available types of predefined color dynamics:

- *Color Blinking* dynamics described above is used to turn the blinking on or off based on the boolean input signal
- *Color Blinking Alert* dynamics can be used to activate color blinking when a value of the input variable goes out of range
- *Color Alert* dynamics changes an object's color when the input value goes out of range.
- *Color Threshold* changes an object's color based on the defined thresholds for the input variable

Predefined dynamics can be attached to the *Visibility* attribute to change object visibility based on the input value, as well as blink the object.

Predefined dynamics can be also attached to the *TextString* attribute of a text object. For example, the *Value Display* dynamics can be attached to the *TextString* attribute to display a numerical value of an input variable using a specified format.

Using Stock Transformations

While predefined dynamics cover most common attribute dynamics, an application developer may need to use stock transformations to add elaborate custom dynamic behavior to an object's attributes. Stock transformations provide greater control over all aspects of the dynamics, and are, in fact, used as building blocks for predefined dynamics.

The following sections provide simple examples of using stock transformations. Although the dynamic behavior described below may be achieved using predefined dynamics, this section concentrates on how to implement various attribute dynamics using stock transformations.

Complex dynamic actions can be constructed as a collection of several stock transformations wired together to implement custom dynamic behavior.

Color dynamics

In addition to setting the RGB value of objects' color attributes using the resource mechanism, color dynamics may be used to allow selecting a color from a list of colors.

If the *Properties* dialog for the *Poly1* object is not open, select the *Poly1* polygon and open the *Properties* dialog for it.

Select the ellipsis button  next to the *FillColor* attribute to activate the *Attribute* dialog.

Select the *Add Dynamics* button at the bottom of this *Attribute* dialog. This will display the list of available dynamics on the right of the dialog. Select *Stock Dynamics* at the end of the list to display a list of available stock transformations. Select the *List* dynamics button, which attaches a list of color values to the attribute and brings the *Edit Dynamics* dialog showing the *List* dynamics attached to the *Fill Color* attribute object.

The *Edit Dynamics* dialog may later be invoked by selecting the Dynamic *Edit* button from the *Attribute* dialog or by clicking on the *X* button on the right of the *FillColor* attribute on the object's *Properties* dialog.

Enter *ColorIndex* as a name for the dynamics' *Value Index* parameter in the right-most field. This resource name will be used to change the color displayed. The value of 0 will display the first color, the value of 1 will display the second color, and so on.

Select the ellipsis button  next to the *List of Values* to display the list of colors. Select the first item in the list and define it's color by selecting a color from the palette, then select the second item and define its color as well.

Click on the *Add* button at the bottom of the *List* dialog to add the third color value and define its color.

Try changing the value of the *Value Index* parameter from 0 to 1 and 2 and notice the colors changing according to the colors in the list.

Animate the drawing with the following script:

```
$datagen -sleep 0.2 d 0 3 $Widget/Poly1/ColorIndex
```

where *-sleep 0.2* is used to slow down the animation by pausing for 0.2 sec. after each update.

Note: the color of the attribute itself is added to the color from the list, so the attribute value is set to black (RGB value of [0,0,0]) when the list dynamics is applied to avoid interference.

Blinking

GLG objects may have a blinking or simple animation functionality added in the GLG Builder, so that the "blinking effect" or animation is incorporated into the drawing and achieved with no programming.

It is accomplished by adding a timer transformation to an object's attribute which needs to be animated. To achieve blinking effect, the value of the attribute needs to be toggled between two values representing the OFF and ON state. Timer transformation can be added only to an attribute of a *D* type. For example, it may be attached to object's *Visibility* attribute to toggle the object's visibility on and off. To implement color blinking, the timer transformation can be attached to the *ValueIndex* attribute of the color list transformation attached to an object's color.

Let's add timer transformation to the *ColorIndex* attribute of the *Poly1* object.

Go down into the widget's viewport, select the *Poly1* polygon and bring its *Properties* dialog. Click on *X* button to the right of the *FillColor* attribute. This will bring a dialog for editing the *List* transformation attached to the polygon's *FillColor*. Click on the ellipsis button next to the *ValueIndex* attribute (which is named *ColorIndex*).

In the *Attribute* dialog of the *ColorIndex* attribute, select the *Add Dynamics* button and select *Timer*. This will add a Timer transformation to the attribute and pop up a dialog for editing the transformation's attributes.

To alternate between 3 colors of the color list transformation, set the *Period* attribute to 3. Name the *Interval* attribute *TimeInterval* and set its value to 0.5 (time interval is defined in seconds). Set *MaxValue* to 2, since we want the *ColorIndex* attribute to alternate between values 0, 1 and 2.

Name the *Enabled* attribute *AlarmEnabled*, so that it can be accessed as a resource if needed. When the attribute is set to 1, the timer will go off automatically with a specified timer interval, and the object will "blink" at run time with no programming involved. The blinking effect will be achieved by alternating between 3 colors in this case (to alternate between two colors, set *Period*=2 and *MaxValue*=1). To turn off the timer, the *AlarmEnabled* resource may be set 0.

Let's animate the drawing by clicking on the *Start* toolbar button. Erase the animation command, since no resources need to be animated (the timer goes off automatically), and click *OK*. The polygon will change its color every 0.5 seconds to indicate an alarm state.

Quit the prototype.

Blinking effect also may be achieved by adding a Timer transformation to the *Visibility* attribute of an object. As an example, let's use a text object and add blinking functionality to it by toggling its visibility.

Create a fixed text object by clicking on the *Fixed Text* button in the *Object Palette*. Click in the drawing to define its anchor point and enter the string to be displayed, for example *ALARM*, then click on *OK* to finish.

Bring the *Properties* dialog for the text object. Name the object *Label* and set *HasResources=YES*.

Click on the ellipsis button next to the *Visibility* attribute to pop up its *Visibility Attribute* dialog. Click on the *Add Dynamics* button and select *Timer* transformation.

When the timer is disabled, the value of the text object's *Visibility* attribute will be set to the timer's *MinValue* attribute. To make text object visible when the timer is disabled, set *MinValue*=1 and *MaxValue*=0.

Set the timer's *Interval* attribute to 0.3 to define 0.3 sec. time interval. Name the *Enabled* attribute *AlarmEnabled*. When it is set to 1, the timer will go off and the text will blink. The *Label/AlarmEnabled* resource can be set programmatically to enable or disable text object's blinking.

Let's animate the drawing again by clicking on the *Start* toolbar button. A polygon object will blink by alternating its color, and the text object will blink by alternating its visibility. Quit the prototype.

The timer transformation may also be used to animate attributes of objects or object's transformations. For example, a timer transformation may be connected to an angle attribute of the rotate transformation to continuously rotate the object the transformation is attached to.

Text dynamics

The simplest way to implement text dynamics is by setting the string attribute of the text object directly from a program. If a numerical or formatted value has to be displayed, GLG also provides a way to format the numerical value right in the drawing instead of coding it in the program.

Numerical text dynamics

This type of dynamics is used to display formatted numbers. Let's create a text object to exercise this type of dynamics. We'll use the *Fixed Text* in this section, but the dynamics may be applied to any type of a GLG text object.

Go down inside the widget viewport. Select *Fixed Text*  from the *Object Palette*, click in the drawing to define text's position and enter the text string. Bring up the *Properties* dialog and select the ellipsis button  next to the *TextString* attribute to bring up the *Attribute* dialog for the string attribute.

Select the *Add Dynamics* button from the *Attribute* dialog to list the available text dynamics.

Select the *Stock Dynamics* option and click on the *Format D* button to add numerical format dynamics. This will activate the *Edit Dynamics* dialog, showing the format dynamics attached to the text object's *String* attribute.

Name the *Data* parameter *Value1* by entering the name in the right-most field. This resource name may now be used to supply data for display. Try entering a different numerical value for the *Data* parameter and see how it changes in the drawing.

The *Format* parameter is a C language double format specification used to format the output. It must use some form of the *%f, %g* or *%e* C format for double values. The number after the decimal point defines the number of digits after the decimal point displayed in the output. You may also add other format symbols, for example try changing format to read *"Value1 = %.3f"*. The *Format Type* attribute may be used to display the value as an integer using one of the integer formats: *%d, %x* or *%o*. The type of the format specification must match the setting of *Format Type* attribute.

Formatted text dynamics

The *FormatS* dynamics is similar to the numerical text dynamics described above. The only differences are that the *Data* parameter is a string instead of a double numerical value, and the format uses the "*%s"* C format specification to format the output.

As an example, try attaching a formatted text dynamics to the Text String attribute of another text object and name it's *Data* parameter *string1*. Use *"string1 = %s"* as the format and *"Hello, world!"* as the value of the *Data* attribute.

List of strings dynamics

The list dynamics used above for colors may also be applied to the *TextString* attribute of the text object. In this case the list will contain strings to be displayed, and the *Value Index* parameter of the list dynamics will control which string is displayed.

Changing attribute range

The *Range Conversion* transformation may be used to change the range of an object attribute or dynamics' parameter to adjust to the range of a variable used in a program. For example, it may be used to change the range of the controlling variable of the rotate dynamics attached to the *Poly1* polygon created earlier.

By default, when *Factor* changes in the range [0; 1], the object rotates in the range [0; *ZAngle*]. For example, if *ZAngle*=180, setting *Factor*=1 rotates the object by 180 degrees; setting *Factor*=0.5 rotates the object by 90 degrees.

Let's try changing the range so that the object rotates between 0 and *Z Angle* when the *Factor* parameter changes between a low and high range different from [0; 1].

Set *Z Angle* to 180. Our goal is to be able to supply a value for the *Factor* in the range [0; 10] instead of [0; 1] to rotate the object between 0 and 180 degrees.

Go down inside the widget viewport. Select the *Poly1* object, bring up the *Edit Dynamics* dialog (use the *Edit Dynamics*  toolbar button), make sure the *Rotate* dynamics is selected. Note that in our example, *Factor* attribute is named *RotateVar*.

Click on the ellipsis button  next to the *Factor*, then click on *Add Dynamics* to display a list of available transformations.

Select *Range Conversion* to attach a range transformation. The attached *Range Conversion* transformation will appear in the *Edit Dynamics* dialog, with a list of transformation's parameters displayed in the middle of the dialog.

InLow and *InHigh* parameters represent the range of the incoming data. *OutLow* and *OutHigh* represent the normalized range of the transformation's output value and should remain 0 and 1.

InputValue represents the incoming data value. The *Range Conversion* transformation takes the value of the *InputValue* attribute as input and converts it from the *InLow/InHigh* range to the *OutLow/OutHigh* range. For example, set *InLow* to 0, and set *InHigh* to 10. Set *InputValue* to 5; it will make the object rotate by 90 degrees. Setting *InputValue* to 0 will return the object into its original position; setting *InputValue* to 10 will rotate the object by 180 degrees.

Notice that the resource name *RotateVar* is now assigned to the *InputValue* parameter of the *Range Conversion* transformation, as opposed to the *Factor* attribute of the *Rotate* transformation.

To expose transformation's parameters as resources, the parameters must be named. Using the rightmost text entry box in each parameter's row (in the *Name* column), name *InLow* parameter *Low*, and name *InHigh* as *High*.

Click on the *Back* button, to go back to the *Rotate* dynamics screen. The *Factor* attribute is now unnamed, as the *RotateVar* is now the name of the *InputValue* attribute of the *Range Conversion* transformation. The value of the *Factor* attribute is desensitized and cannot be changed directly, since the value of the *Factor* is now controlled by the *Range Conversion* transformation. The *x*

button to the right of the *Factor* row indicates that *Factor* has a transformation attached. Click on the *x* button to display the *Range Conversion* transformation. Click on the *Back* button to go back to the *Rotate* dynamics.

With the object being selected, click on the *Resources* toolbar button to display object's resources. Since the object's *HasResources* flag is set, *Low*, *High* and *RotateVar* resources will appear in the Resource Browser as resources of *Poly1*. *Low* and *High* resources can be set at run-time to define the low and high range of the incoming data, and the *RotateVar* resource can be used to supply real-time data values to animate the object.

To return to the object's transformations at a later time, select the object, bring *Properties* dialog, click on *Dynamics Edit* button to display *Rotate Z* dynamics. To access *Range Conversion* attached to the *Factor*, click on the *x* button to the right of the *Factor*, or click on the ellipsis button  next to the *Factor* and click on the *Dynamics Edit* button in the activated *Attribute* dialog.

The range transformation may be attached to any scalar (D) attribute of any object.

Editing Control Points and Attaching Control Point Dynamics

You can adjust control points by moving them with the mouse. You can also *Shift*-click on the control point (click on the point with the left mouse button while pressing the *Shift* key) to bring up the *Control Point* dialog for fine tuning the control point values. *Shift*-click is also convenient for selecting one out of several closely positioned control points.

You can use the *Control Point* dialog to enter the X, Y and Z coordinate values of the control point (in the -1000 to +1000 range for the default coordinate mapping).

You can also use the directional buttons to fine-tune the control point position, moving it by one or more pixels in a selected direction.

You can *Shift*-click on the *Move Control Point* (a special  point in the center of the object) to move the object using directional buttons. You can also *Shift*-click on the *Rotate Point* (a special point on the right side of the *Resize Box*) to rotate the object by a defined angle using directional buttons.

You can add geometrical dynamics to individual control points by selecting the *Add Dynamics* button in the *Control Point* dialog, and then proceeding in the same way as when attaching geometrical dynamics to objects.

To move several control points uniformly, use the constrained transformations described in the *Using Constrained Dynamics and Marked Transformations* section.

Adding Extended Rendering Attributes

A rendering object may be attached to GLG graphical primitives (polygon, arc, etc.) to define an optional set of extended rendering attributes. These attributes are not part of the object by default for the sake of efficiency, and are added only to the objects that truly need them. The extended rendering attributes include attributes that define and control several types of gradient fill, cast shadows, arrowheads and fill dynamics.

To add rendering attributes to an object, select the object, display its Properties dialog, then click on the Add Rendering button at the end of the properties list. This will add rendering to the selected object and display the properties of the rendering object. For example, create a filled polygon, set its *FillColor* to white, display its Properties dialog and add rendering to it.

If the selected object has already had rendering added, the *Edit Rendering* button will be displayed, allowing the user to access the rendering attributes for editing.

To delete the rendering, press the *Delete Rendering* button at the end of the Rendering Properties list.

To return to the object attributes without deleting Rendering Attributes, press the *Previous* button at the bottom of the *Properties* dialog.

Gradient Fill

To add a linear gradient fill to the polygon created above, change the *Gradient Type* from NONE to ACYCLIC LINEAR in the *Rendering Properties* dialog. The object will be rendered with a gradient fill, changing its color from its original *FillColor* to the *GradientColor*.

The *GradientColor* attribute defines the second color for the gradient fill. Press the ellipsis button  next to the *GradientColor* to display a color palette for selecting it.

Change the *GradientType* to CYCLIC LINEAR and notice the change in the gradient fill: it now cycles from the object's *FillColor* to the *GradientColor* and back to the *FillColor*.

The *GradientAngle* attribute controls the angle of the gradient, measured counter clock-wise relative to the X axis. Change the angle to 90 to render a vertical gradient fill.

Change the *GradientType* to CONICAL and then SPHERICAL to see the corresponding gradient fills. Finally, change the *GradientType* from SPHERICAL to INVERSED SHPERICAL and notice the change in the gradient fill colors: instead of changing from the object's *FillColor* to the *GradientColor*, the color change is inversed.

The *GradientLength* attribute controls area of the object rendered using the gradient fill. Change the *GradientLength* to 0.5 to see only half of the object area filled with the corresponding gradient.

The *GradientCenter* attribute defines the gradient center in relative coordinates. Change the *GradientCenter* to 0,0,0 to have the gradient centered at the lower left corner of the object's bounding box. Setting the attribute's value to 1,1,1 centers the gradient at the top right corner of the object's bounding box.

The *GradientResolution* defines the number of polygon segments used to render the gradient fill. On non-True Color systems, due to the lack of available colors, the actual number of polygons used may be less than the specified value.

Fill Dynamics

The *FillAmount* attribute is used to implement the fill dynamics commonly encountered in process control and other drawings showing tanks filled with liquid substances. A changing substance level may be implemented by adding rendering attributes and changing the *FillAmount* of the object representing the tank.

Try changing the value of the *FillAmount* of a filled polygon with rendering attributes. If the value is 1, the whole object is drawn. If the value is less then 1 (0.5, for example), only a portion of the object's fill will be drawn, as defined by the fill amount. If the value is 0, the fill is not drawn (i.e. tank is empty). Notice that the object's edge is not affected by the value of the *FillAmount*, and that the object's fill type must be either FILL or FILL AND EDGE in order to use fill dynamics.

The *FillAmount* attribute may be named for easy access (press the ellipsis button  next to the *FillAmount* and enter the name in the *Attribute* dialog). The name of the *FillAmount* will appear in the resource browser together with the other object resources.

The *FillDirection* attribute defines the direction of the fill dynamics. You may select one of the 4 preset values: UP, DOWN, LEFT or RIGHT, or define an arbitrary angle. To define an arbitrary angle, press the ellipsis button  next to the *FillDirection* and enter the value of the fill angle. The angle is measured counter clock-wise, relative to the X axis (for example, an angle of 90 corresponds to the UP fill direction).

Shadows and Arrowheads

The *ShadowOffset* attribute controls the offset of the cast shadow. The offset is in screen pixels, and if the offset's value is not 0 in the X or Y direction, a shadow is rendered, as defined by the *ShadowColor* attribute. Press the ellipsis button  next to the *ShadowColor* to display a color palette for selecting the color.

The *ArrowType* attribute defines the type (LINES or FILLED) and position (one of the ends, both ends, or the middle) of arrowheads drawn on a polygon, arc, and other polygon-like objects. The arrowheads are drawn when the *ArrowType* differs from *NONE*. Try selecting different ArrowType values and observe the resulting arrowheads. The *ArrowShape* attribute allows the user to define a custom arrow shape by specifying the X and Y extent of the arrowhead.

Text Boxes

The text object allows the user to define an optional text box around the text by attaching the Box Attributes object. The Box Attributes define the attributes of an outline or filled box drawn around the text.

Try creating a text object and adding Box Attributes by clicking on the ellipsis button next to the *Add Box Attributes* label at the end of the text's *Properties* dialog. An outline drawn around the text will appear. Change the *FillType* from EDGE to FILL & EDGE to display a filled box, then select the FillColor of the box (click on the FillColor's ellipsis button to display a color palette). The *BoxOffset* attribute defines an offset in pixels between the text and the edge of the box.

To delete the Box Attributes, click on the *Delete Box Attributes* button at the bottom of the Box Attributes' properties.

To return to the text object's attributes without deleting the Box Attributes, press the *Previous* button at the bottom of the *Properties* dialog.

Using permanent groups

A *group* object provides a convenient way to hold several objects together, so that they can be saved, moved or edited as one entity.

Creating a group

To create a group, select the *Group* icon from the Object Palette, then click and drag the mouse to define a rectangular area. A new group will be created and all objects that are either completely or partially inside the defined rectangle will be placed inside that group.

When the group is selected, the control points of all objects in the group are displayed. Also, the group's control points don't have a cross in the middle.

Editing Individual Objects in a Group

To get access to individual objects in the group, go inside the group in the same way you got used to with the viewport object. Select the group and click on the *Hierarchy Down* button to zoom into the group. After this you'll be able to select and edit individual objects inside the group. Go back up when finished editing.

You can also select individual objects in the group in the context of the drawing without traversing down. Click on *Select Next* in the group's *Properties* dialog and selecting the object in the group you want to edit. After editing an object, simply select the next one with the mouse. Select an object outside of the group to finish editing group objects.

When several groups are nested, you may use *Select Bottom* from the group's *Properties* dialog to select an object on the bottom of the group's hierarchy.

Select Next and *Select Bottom* are also available as toolbar buttons. When a permanent group is selected, *Ctrl-Shift*-click on an object inside the group also works as a shortcut for the *SelectNext* operation.

Editing All Objects in a Group

Groups provide a convenient way to edit many objects at once. To edit multiple objects in an existing group, select the group, click on the *Edit All* button in the group's *Properties* dialog or use the *Edit All (First)* option in the *Traverse* or *Arrange* menus. The properties to be edited will be determined by the type of the first object in the group. As you edit any of the attributes, all objects in the group that have that attribute will be changed. To return to the properties of the group, click on the *Prev* button at the bottom of the *Properties* dialog.

A group may contain objects of different types, such as polygons or text objects, in which case the *Edit All (Select)* option allows you to select a set of properties to edit by clicking on an object in the group. The type of the selected object will define the set of attributes to edit.

For example, to edit *Anchoring* of all text objects in the group, click on *Edit All (Select)* and select any text object in the group with the mouse to edit text attributes. To edit the *Line Width* of all polygons in the group, use *Edit All (Select)* and select a polygon to define the polygon attribute set.

If you constrain an attribute (see the *Using Constraints* section below) while editing all objects in the group, the corresponding attribute of all the objects in the group will be constrained.

If you want to edit multiple objects that do not belong to a group together, create a temporary group as described in the the *Multiple Selection* section on page 16. The temporary group will be destroyed automatically when unselected.

Adding and Deleting Objects from a Group

You can add or delete objects from a group by using the *Add To Group* and *Delete From Group* toolbar buttons.

To add to a group, select the group, click on the *Add To Group*  toolbar button and select objects in the *Drawing Area* you want to add. Press the *Escape* key to finish.

To delete objects from the group, select the group, click on the *Delete From Group*  toolbar button and select the objects in the group you want to delete. The objects will be deleted from the group and added to the *Drawing Area*.

You can also zoom down into the group to manipulate group objects one by one.

Exploding Permanent Groups

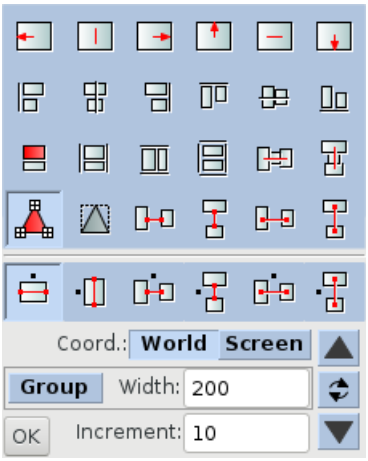
To release the objects from the group, explode the group by using the *Explode Object*  toolbar button or *Arrange, Explode, Explode Object* from the main menu.

Explode operation may be applied to other objects as well, such as circles, arcs or series, but keep in mind that *Explode* action can't be reversed when executed.

Object Layout and Alignment

If you need to align or layout a group of objects, *Layout Toolbox* provides a point and click interface for such functionality. Examples of layout and alignment operations include aligning several objects horizontally or vertically, setting the same width for several objects, positioning objects within a defined distance from each other, or distributing the objects evenly across the defined extent in a horizontal or vertical direction.

Click on the *Layout Toolbox* button in the toolbar to activate a *Layout Toolbox*. Alternatively, the *Layout Toolbox* can be activated using *Layout*, *Layout Toolbox* option in the main menu. The following picture shows the *Layout Toolbox:*



The top of the *Layout Toolbox* contains icons for actions that do not require a value, such as aligning objects or making them the same width or height. The bottom of the toolbox contains controls for actions that require a value, such as setting an object's width or height to a specified value.

To experiment with the *Layout Toolbox*, create three rectangles using the *Filled Rectangle* button and position them horizontally, one next to another. Select all rectangles by clicking and dragging the mouse to define a rectangular area that encloses all three rectangles. This will create a temporary group containing all three rectangles.

Click on the *Align Bottom* button in the *Layout Toolbox* and notice that the bottom of all rectangles will be aligned.

The *Select Anchor Object* button in the *Layout Toolbox* may be used to define an anchor object. If an anchor is selected, its dimensions are used to align other objects. Click on the *Select Anchor Object* button, then click on one of the rectangles to define it as an anchor.

If the anchor is not defined, one of the objects in the group will be chosen as an anchor automatically. Once the anchor is defined, it will be persistent for all layout operations until another anchor is chosen or the objects are unselected.

Click on the *Set Same Height* button to set the height of all rectangles to the height of the anchor rectangle.

The icons in the upper half of the toolbox perform actions when you click on them, while the icons in the lower half of the dialog define actions to be performed when a numerical value is entered.

For example, follow these steps to position the rectangles with an even horizontal space between them:

- Click on the *Set Horizontal Space*  button to select the operation to be performed. Notice that the label of the text edit box at the bottom of the *Layout Toolbox* displays the selected operation and is set to *Horiz.Space.*
- Enter 50 as the value in the *Horiz. Space* text box and press *Enter*. This will position all rectangles to be within 50 world coordinates from each other in the horizontal direction.
- Use the Up and Down buttons to increase or decrease the space between the rectangles using the specified *Increment*.

The *Coord: World* / *Screen* control toggles between world and screen coordinates.

Press the *Escape* key to unselect the rectangles.

Creating Layers of Objects

A layer is a collection of objects whose visibility may be turned **ON** or **OFF**. Layers are usually used to display collections of objects or icons on maps, schematic diagrams, etc.

A layer of objects may be created by placing several objects in a permanent group. The group's visibility attribute is then used to make the layer visible or invisible by setting just one visibility attribute (of the group). Objects in the layer may use attribute constraints described in the previous section to constrain some attributes to have the same value. For example, if you constrain the color attributes of all the layer's objects, changing one color resource will change the color of all objects. Different layers may use different colors, line widths, etc., to accent the layer.

2. Advanced Features of the GLG Builder

Using Constraints

Constraints may be used to constrain the value of an object attribute to an attribute of another object. This simplifies maintaining or animating collections of objects.

For example, imagine a drawing or a car that uses different polygon objects to render different parts of the body. If the color attributes of all the polygons are constrained, you can change the color of just one polygon object and the color of the whole car will change.

In more complex cases, parameters of dynamics may be constrained to change synchronously as will be shown in the *Using Constrained Dynamics and Marked Transformations* section.

The **constraints tracing** option described in the *Constraints Tracing* section on page 316 of the *GLG User's Guide and Builder Reference Manual* may be useful for debugging constraints defined in the drawing.

Constraining Object Attributes

To exercise attribute constraints, start with a new widget (use *File, New Widget* from the main menu). Create three polygons in the drawing and name them *Poly1, Poly2* and *Poly3*. Select *Poly1*, bring up it's *Properties* dialog and select the ellipsis button  next to it's *Fill Color* attribute to activate the *Attribute* dialog.

Select the *Constrain* button on the left of the *Attribute* dialog and select the *Poly2* polygon in the drawing with the mouse: this will constrain the *Fill Color* attribute of the currently selected polygon to the *Fill Color* attribute of the polygon you selected with the mouse.

Try changing the color by selecting a new color from the color palette: the color of both polygons will change.

This way of constraining works for constraining attributes of the same type. It will not work for example, for constraining *Fill Color* of one polygon to the *Edge Color* of another polygon. To do this, the Builder has features to mark attributes and later use marked attributes.

Click on the *Mark* button in the *Attribute* dialog to mark the *Fill Color* attribute of *Poly1*, which is still selected (in editions of the Builder other than Basic, you also have to select *Mark 0* from the list since more then one mark is available).

Now select *Poly3*, bring up the *Attribute* dialog for its *Edge Color* attribute and click *Constrain*. To constrain the edge color to the attribute that was previously marked, select the *Use Marked* button from the list on the right of the *Attribute* dialog (another option is to constrain to a resource by using the *Resource Browser*).

If you now select a new color from the palette, the *Edge Color* of *Poly3* and the *Fill Color* of the other two polygons will change simultaneously.

The marking feature may also be used to constrain parameters of object's dynamics to change in unison.

The object's control points (other then the move point) may be constrained as well. *Shift*-click on a control point to bring up the *Control Point* dialog, click on *Constrain* button and select a control point of some object in the drawing to constrain to. If you now move the control point, both points will move.

The *Attribute* dialog also has the *Merge* button. If several attributes are constrained, the *Constrain* button unconstrains from a previous place and constrains to a new place, while the *Merge* button constrains the attribute itself as well as all other attributes that were constrained to it to a new place.

If you're not sure, use the *Merge* button as a first choice to preserve any existing constraints of complex objects.

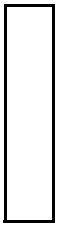
Using Constrained Dynamics and Marked Transformations

Constrained Dynamics Example

Constrained dynamics is convenient when you need to move several objects together. In case of geometrical objects, the simplest way to move them together is to place them in a group and then attach dynamics to the group. However, if you want to move two control points in the same way, constrained dynamics provides the best solution.

For example, let's try to create an animated object that resembles a bar of a bar graph.

Start with a new widget (use *File, New Widget*). Create a filled, rectangular shaped polygon by clicking on the *Filled Polygon*  icon and selecting 4 points in the drawing as shown in the picture:



Don't define the fifth point: it will be completed by the polygon when you finish entering points by pressing the right or middle mouse button.

Now we are going to animate the top two points of the polygon to move up and down together. *Shift*-click on the first top point to bring up the *Control Point* dialog and select *Add Dynamics* button to activate the *Add Dynamics* dialog. Select the *Move* transformation type, click on the *Move Vector In Drawing* button, and select two points in the drawing to define a vertical moving distance. Enter *Height* as the *Variable Name* and hit *Apply.*

This will attach the *Move* dynamics to the control point and will bring up the *Edit Dynamics* dialog showing the parameters of the attached transformation. *Height* should appear as the name of the dynamics' controlling *Factor*. Try changing the factor from 0 to 0.3, for example, and notice the point moving in the drawing.

Select the *Mark Object* button from the *Edit Dynamics* dialog to mark the currently selected dynamics for later use (the *Mark List* button marks all dynamics attached). Then close the *Edit Dynamics* dialog, *Shift*-click on the second top point and select the *Add Dynamics* button.

But this time we are not going to attach a *Move* dynamics. Instead, we'll reuse the same dynamics that we just marked. Change the transformation type from *Move* to *Use Marked*, and select the *Constrained* clone type. This means that a copy of the marked *Move* dynamics will be attached, and all parameters of this copy will be constrained to the parameters of the marked dynamics.

Hit *Apply* to attach the dynamics and try changing the value of the controlling *Factor* (named *Height*) from 0.3 to 0.5: both points will move the same distance.

Let's adjust the initial position of the moving points to be at the bottom of the bar. Set the *Height* parameter to 0, close the *Edit Dynamics* dialog and move the top points down to coincide with the bottom points when the height equals 0.

Now prototype the drawing with the following animation script:

```
$datagen -sin d 0 1 $Widget/Height
```

The script will change the value of the *Height* resource from 0 to 1, resulting in the two top points moving up and down by a defined distance.

The above example attaches move dynamics to two points of a polygon to demonstrate how to move several objects together. This particular dynamic can also be accomplished by using a rectangle object with dynamics attached to one of its points.

Using Marked Transformations

Marking and then using marked transformations (as described in the above example) is a convenient way to add the same transformation to several objects. In the transformation dialog, one transformation or a whole list may be marked for reuse. Selecting the *Use Marked* choice of transformation types will attach one or more marked transformations to the object.

The *Clone Type* option controls the constraining of the cloned object's attributes: if *Full Clone* is selected, a fully independent copy of the transformation is attached, *Constrained Clone* attaches a constrained copy with all attributes of the copy (or copies) constrained to the corresponding attributes of the original transformation(s). The *Weak Clone* constrains only the attributes with *Global* flag set to GLOBAL, and *Strong Clone* constrains attributes with the GLOBAL or SEMIGLOBAL settings of the *Global* flag.

The Second Flavor of the Fill Dynamics

The constrained control points dynamics described in the previous section may be used to create a Fill Dynamics similar to the one described in the *Adding Extended Rendering Attributes* chapter.

While using the *FillAmount* rendering attribute is the easiest way to implement fill dynamics, the fill dynamics implemented using control points' dynamics allows a few additional features: it affects not only the fill but also the edge of the object, and it rotates with the object when the object is rotated. For example, if *rotate* dynamics is attached to the object, the fill will rotate with the object instead of continuing to fill vertically or horizontally. The fill dynamics implemented with the FillAmount may be rotated only by changing the angle of the FillDirection attribute.

Defining Object Tooltips

The GLG drawing supports integrated object tooltips. If an object (or any of its parents) have a tooltip action attached, the value of the action's *Tooltip* attribute will be used to display a tooltip when the mouse hovers over the object. If the object is a group, the tooltip will be displayed every time the mouse moves over any of the group's subobjects.

To attach a tooltip to an object, create an object and select *Object, Add Tooltip* from the main menu (Enterprise Edition of the Builder is required to add the tooltip). This will attach a tooltip action to the object and display a dialog for editing the action's attributes. Enter the tooltip string in the text box next to the action's *Tooltip* attribute. To enable tooltips, set the *ProcessMouse* attribute of the viewport that contains the object to a value that includes the *Tooltip* mask. To try the tooltip in the prototyping mode of the Builder, press the *Start* toolbar button and enter an empty run command. Move the mouse over the object and wait a fraction of a second to see the tooltip. The tooltip disappears when the mouse is moved away from the object. Press the *Stop* toolbar button to return to the edit mode.

At run time, the tooltip is handled by the Toolkit automatically, with no program actions required. Refer to the *Integrated Tooltips* chapter of the *GLG User's Guide and Builder Reference Manual* for more details.

Using MouseOver Highlight and MouseClick Feedback

The GLG drawing can highlight objects in the drawing when the mouse is moved over them. This is done by attaching an action with *ActionType=TRACE_STATE* and *Trigger=MOUSE_OVER* to an object. The action's *State* attribute will be set to 1 when the mouse moves over the object and reset to 0 when the mouse moves away. The *State* attribute may be constrained to an object's attribute, such as a polygon's *LineWidth,* or to an attribute of a dynamics, such as *Value Index* of the *List* dynamics attached to the object's *FillColor*, to modify the object's appearance when the mouse moves over the object.

If the action's *Trigger* attribute is set to *MOUSE_CLICK*, the value of the action's *State* attribute will be set to 1 when object is clicked with the mouse, and reset back to 0 when the mouse button is released. If *ActionType*=TOGGLE_STATE, the value of the State attribute is toggled between 0 and 1 every type the mouse clicks on the object.

To try the *MouseOver* highlight, create a filled polygon, name it, set its *FillType* attribute to *FILL & EDGE*. Press the ellipses button next to the *LineWidth* attribute to display the *Attribute* dialog for *LineWidth*, then press the *Add Dynamics* button.

Select the *Stock Dynamics* option and select the *List* button to add list dynamics to the *LineWidth*. The *Edit Dynamics* dialog will be displayed.

In the *Edit Dynamics* dialog, press the ellipses button next to the *Value Index* attribute, then click on *Mark* and *Mark0* buttons in the *Attribute* dialog to mark the *Value Index* attribute for reuse.

In the *Edit Dynamics* dialog, click on the *List of Values* button to edit the list items. Select the first list item and set its value 1, then select the second item and set its value to 3 to alternate between the thin and thick lines.

With the polygon still selected, use the *Object, Actions, Add Mouse Feedback* menu option to add a mouse feedback action with *ActionType=TRACE_STATE* to the polygon, then change the action's *Trigger* to *MOUSE_OVER*.

Press the ellipses button next to the *State* attribute, then use *Constrain All* and *Use Marked* buttons in the Attribute dialog to constrain *State* to the previously marked *Value Index* attribute (the *Mark0* button may need to be selected if more than one attribute was previously marked).

Set the *ProcessMouse* attribute of the viewport containing the object to *Move & Click*, then start the prototyping mode by pressing the *Start* toolbar button and entering an empty run command. The polygon will be highlighted by changing its line width every time the mouse moves over it. Press the *Stop* toolbar button to return to the edit mode.

Alternating the *LineWidth* attribute is just one of the ways to highlight the object. Various other types of highlighting may be implemented by using different attributes and transformation types. The following examples use different transformation type to implement a different form of visual feedback.

To define the *MouseClick* feedback for the object, select the polygon object used in the previous example and select the *Add Dynamics* toolbar button. Select the *Move* transformation type in the *Add Dynamics* dialog, set the *Start Point* to "0 0 0" and *End Point* to "50 50 0", then press the *OK* button to attach the move transformation to the object.

In the *Edit Dynamics* dialog, press the ellipses button next to the *Factor* attribute, then click on *Mark* and *Mark0* buttons in the *Attribute* dialog to mark the attribute for reuse.

With the polygon still selected, use the *Object, Actions, Add Mouse Feedback* menu option to add a mouse feedback action with *ActionType=TRACE_STATE* and *Trigger=MOUSE_CLICK* to the polygon.

Press the ellipses button next to the *State* attribute, then use *Constrain All* and *Use Marked* buttons in the Attribute dialog to constrain *State* to the previously marked *Factor* attribute (the *Mark0* button may need to be selected if more than one attribute was previously marked).

The *ProcessMouse* attribute of the object's viewport already includes the *Click* mask, so you can start the prototyping mode (select *Start* toolbar button and enter an empty run command).

Try clicking the object with the mouse and notice it moving a little bit diagonally and then returning back when the mouse button is released. The *MouseOver* feedback is also displayed when the mouse moves over the object. Press the *Stop* toolbar button to return to the edit mode.

Select the polygon object again, use the *Object, Actions, Edit Actions* menu option to display the list of actions attached to the polygon, then select the second action (the one with Trigger=MOUSE_CLICK) and change its *ActionType* to *TOGGLE_STATE*. Run the drawing in the prototyping mode again and notice that now the object alternates its position every time you click on it.

At run time, the mouse feedback actions are handled automatically, with no program code required. Refer to the *MouseOver Highlight and Cursor Change* and the *MouseClick Feedback and Toggle* chapters of the *GLG User's Guide and Builder Reference Manual* for more details.

Attaching Custom Events and Commands

An integrated action may be attached to an object in the GLG drawing at design time to define a command or a custom event to be triggered at run time when the object is selected with the mouse, via either *MouseClick* or *MouseOver*. To attach an action to an object, select the object and use one of the options of the *Object, Actions* menu of the Enterprise Edition of the Builder, or the HMI Configurator.

The action's attributes specify the type of action, its activation conditions (such as *MouseOver* or *MouseClick* events), as well as the mouse button used to trigger the action. The action may also contain additional parameters needed to execute the command associated with the action.

The viewport's *ProcessMouse* attribute has to include a combination of *Move* and *Click* masks to enable processing of the corresponding mouse events.

Input actions may be attached to the input objects, such as buttons or sliders, to be triggered by specific user interaction, for example changing a value of a slider or clicking on a push button.

At run time, the program's Input callback will be invoked, providing the program with information about the object and action that generated the event. The program will then use this information to process the command associated with the action to implement the application logic.

Refer to the *Integrated MouseOver and MouseClick Actions* chapter on page 243 of the *GLG User's Guide and Builder Reference Manual* and the *Action Object* chapter on page 219 for more details.

Refer to the *GLG Programming Manual* for more details of the Input callback and custom event handling.

Using Editing Focus

The Editing Focus feature provides a shortcut to edit objects inside the viewport without zooming down into it.

To move the editing focus inside the viewport, select the *Set Focus* button from the *Control Panel* on the left and click on the viewport whose objects you want to edit. The editing focus will be temporarily be moved inside the viewport, changing it's border width to indicate the current focus. You can now select and edit objects inside the viewport.

To move the focus to some other viewport, set the focus again. To reset the focus back to the Drawing Area, select the *Main Focus* button.

The focus is very convenient for editing nested viewports without traversing down several hierarchy levels.

Note: You zoom down only if editing focus is set to the Drawing Area. If the focus is moved into some viewport, reset it back to the Drawing Area before zooming down.

Changing Viewport's Font Tables

The fonts used in a drawing are defined by the viewport's Font Table. You can edit the Font Table of any viewport right in the builder to define the number and types of fonts which can be used. The text objects in the drawing use the font type and font size indexes to select a font from the font table.

The Font Tables are used to optimize real-time access to the font rendering engine and prevent rendering a huge number of slightly different fonts that can bring even the powerful CPU to its knees, since GLG drawings may contain hundreds or even thousands of text objects. The future versions of the Toolkit will also provide an alternative way for specifying the font directly as an attribute of a text object for applications with a smaller number of fancy text objects.

The font tables may contain any font existing on the system, True Type or not, proportional or monospaced. The fonts are organized into font families that contain different sizes of the same type of font. These sizes are used to scale the GLG scalable text objects.

To change the font, click the *Screen Attributes* button in the viewport's *Properties* dialog, click on the *Add Font Table* button to add a custom font table and select the ellipsis button next the *Fonts* label to show the attributes of the font table.

You can then define the number of font types and sizes in the font table, as well as select the fonts used. For each font, you can define separate font names to be used in a Linux/Unix, Windows, Java and JavaScript environments. When finished, you can save the font table for later reuse in this and other drawings.

If a drawing contains several nested viewports, you can define the font table only for the top *$Widget* viewport of the drawing and set the rest of the nested viewports to use the default font table. This will cause the nested viewports to inherit the table from the parent *$Widget* viewport.

3. GLG Widgets and Custom Objects

The GLG Toolkit provides collections of pre-built widgets, such as **Real-Time Charts, 2D Graphs, 3D Graphs, Controls, Avionics, Process Control Symbols, Electrical and Electronic Circuit Symbols**, and **Special** widget sets. Possible widget palettes also include custom palettes provided by an OEM vendor.

The **Professional** and **Enterprise** version of the builder also allow building custom interface objects from scratch (refer to the *Widget Input* chapter of the *GLG Programming Reference Manual* for more details).

The Builder's **Custom Objects** palette provides a few samples of graph and control widgets.

Using Custom Object Palette

The *Custom Objects* palette (the *Custom Objects* + icon in the *Object Palette* or *Palettes/Custom Object* option of the main menu) provides a few predefined interface objects that may be used in GLG drawings.

The *Custom Object* palette provides the following interface objects:

Custom Button

A button with custom graphics inside. To edit or replace the graphics, select the button's viewport and go down into it.

The button has a *TooltipString* resource that defines the tooltip string displayed in the Run mode.

Custom Toggle

A toggle with custom graphics inside. To edit or replace the graphics, select the button's viewport and go down into it. The toggle's graphics must have a resource named *OnState*. The toggle changes the value of this resource between 0 and 1 when you click on the toggle in the *Run* mode.

You may delete the *OnState* resource and provide your own. For example, create a toggle from the palette, select it and zoom down into it. Use the *Resources* toolbar button to bring up the *Resource Browser*, select the *OnState* resource and erase its name in the *Attribute* dialog. Close the resource browser.

Now select the moving red part of the toggle, bring up the *Properties* dialog, click on the ellipsis button ... next to it's *Visibility* attribute and name the *Visibility* attribute *OnState* by typing the name into the *Name* field of the *Attribute* dialog.

Go back up (this also resets the drawing after resource names change) and prototype with an empty animation script. When you click on the toggle, it will change the visible state of the switch. This technique can be used to display a custom checkmark in a toggle.

The toggle also has a *TooltipString* resource that defines the tooltip string displayed in the Run mode.

Native Button Object

This is a native windowing system button object. It's behavior depends on the operating system (Linux/Unix or Windows), and also on the programming environment: it will look differently in C/C++, Java and JavaScript. The name of the button is used as it's label. You'll have to reset the drawing with the *Reset*  toolbar button to see the label change.

Same as the custom button, the button has a *TooltipString* resource.

Native Toggle Object

This is a native windowing system toggle object. Same as a native button, it's look and feel depends on the environment it's displayed in. The name of the toggle is used as it's label. You'll have to reset the drawing with the *Reset*  toolbar button to see the label change.

Same as the custom toggle, the toggle has a *TooltipString* resource and a *OnState* resource that indicates it's current state. The *OnState* resource may be set or queried from a program. Some resources in the drawing, for example the *Visibility* attribute of an object in the drawing, may also be constrained to the *OnState* resource, so that the toggle will change the object visibility when activated.

Native Slider Objects

Slider objects (scrollbar on Windows) represent native windowing system sliders or scrollbars. Their behavior depends on the environment the sliders will be displayed in.

The sliders have a *Value* resource that changes from 0 to 1 when the sliders are moved in the run mode. The *Value* resource may be set or queried from a program. Some resources in the drawing, for example a controlling factor of dynamics, may also be constrained to the *Value* resource, so that a slider will animate the dynamics when the slider is moved in the *Run* mode.

3D Objects

3D *Cube* and *Cylinder* objects are built as collection of polygons that render a cubical or cylindrical 3D shape. To see the shape in 3D, you may need to rotate the view using the 3D rotation controls in the *Control Panel* on the left.

The polygons are grouped, select the group and zoom down into it to edit individual polygons or attributes. The color attributes of the polygons are constrained, so by changing the color of one of polygons, all polygons change their color.

The corner points of the polygons are constrained, which makes it easier to edit the shape. For example, select the cube's group, click on the *Select Next* toolbar button and select one polygon of the cube (you can use *Shift*-click when selecting to get a finer control over selecting).

Try moving the selected polygon with the mouse: the rest of the polygons will stretch to stay connected.

Process Control Objects

Two samples of process control objects: a *valve* and a *tank* are provided in the Custom Object palette. Both objects use a container object to encapsulate their sub-objects.

To edit graphical primitives of these models, zoom down into the container using the *Hierarchy Down* button  from the *Control Panel*, select the object, then zoom down once more to go down into the group object.

The valve has *Rotation* dynamics attached to the moving element with the dynamics' controlling *Factor* named *Angle*. When the *Angle* parameter changes from 0 to 1, the valve gradually changes from fully closed to fully open position.

The tank has the constrained move dynamics connected to the two moving control points, same as the control point dynamics described in the *Using Constrained Dynamics and Marked Transformations* section. The dynamics moves both points up or down when its controlling factor named *Level* changes from 0 to 1. Set the *BackgroundColor* resource of the tank to match the background color of the drawing.

Graph and Real-Time Chart Objects

Two graphs: a *Line Graph* and a *Bar Graph* are provided with the Graphics Builder. Additional graph widget sets may be purchased, including Real-Time Charts, as well as 2D and 3D graphs.

Refer to the *Using GLG Real-Time Charts* chapter of this manual for details on how to use resources of GLG Real-Time Charts. Refer to the *Using Legacy GLG 2D and 3D Graph Widgets* chapter for details on how to use resources of GLG graphs.

Adding New Objects to the Custom Object Palette

New objects may be added to the custom objects palette by saving them in the *widgets/custom_objects* directory. Refer to the *Read Directory* section of the *GLG Graphics Builder Menus* chapter for more details.

4. Using GLG Drawings in a Program

To use GLG drawings created with the GLG Builder in a C/C++, Java, C#/.NET, JavaScript or ActiveX environment, a number of GLG containers and APIs are provided:

GLG cross-platform API (C/C++/C#/Java/JavaScript)
GLG .NET Control (C#/.NET)
GLG Bean (Java)
GLG Control or MFC class (C/C++ on Windows)
GLG ActiveX Control (Windows)
GTK Widget (GTK version of the GLG C/C++ library)
GLG Wrapper Widget (legacy X11 version of the GLG C/C++ library)

Each of these containers encapsulates a GLG drawing for use in one of the programming environments.

To use a GLG drawing in Java, C#/.NET or JavaScript, save the drawing using the default ASCII save format to ensure accessed from any hardware platform. Use the default *Save Compressed* option to decrease the size of the drawing file.

Loading a Drawing into a C or C++ Program

To use a drawing with a C/C++ program, it must contain a viewport named *$Widget*. This viewport will be displayed when the drawing is loaded.

The specific method for loading the drawing depends on the programming environment used.

A **C/C++ program** may use the *GlgLoadWidgetFromFile* function or the *LoadWidgetFromFile* C++ method to load a drawing in a platform-independent way. The drawing is then displayed by using the *GlgInitialDraw* function or the *InitialDraw* C++ method.

In a **Windows environment**, either a GLG MFC control class or a Windows custom control may be used to encapsulate a GLG drawing.

On **Linux**, a *GtkGlg* widget may be used to display a drawing, as shown in the example provided in the *integration/gtk3_glg_native directory* of the GLG installation.

A *GlgWrapper* widget may be used to display a drawing in the legacy X11 version of the GLG library. The *XtNglgDrawingFile* resource of the widget is used to specify the drawing to load.

Refer to the *Displaying a GLG Drawing* chapter of the *GLG Programming Guide* for more details.

Loading a Drawing into a Java Program

The Toolkit provides a GLG Bean Java component used to display and animate a GLG drawing in a Java program. To use a drawing in a Java program, the drawing must contain a viewport named *$Widget*. This viewport will be displayed when the drawing is loaded into the GLG Bean.

The GLG Bean provides parameters that specify either a filename or URL of a drawing to be displayed. Setting one of these parameters loads the drawing and displays it in the bean. A GLG Bean also has methods that allow to load the drawing from a file or a URL on demand under the program's control.

To load the drawing into the **GLG Bean**, set one of the following bean properties: *DrawingName, DrawingFile* or *DrawingURL.* The *DrawingName* property may be set to either a filename or a URL.

To load the drawing into a **Java program**, you can also use the *LoadWidget* method of the *GlgObject* class or the *LoadWidget* method of the GLG Java Bean.

Loading a Drawing into a C#/.NET Program or GLG .NET Control

To use a drawing with a C#/.NET program and GLG .NET Control, it must contain a viewport named *$Widget*. This viewport will be displayed when the drawing is loaded.

GlgControl has properties that define the drawing file to be displayed in the control. Setting one of these properties loads the drawing and displays it in the control's window. The control also has methods that allow to load the drawing from a file on demand under the program's control.

To load the drawing into the ***GlgControl***, set its *DrawingFile* property to load and display the drawing from a file. Use the *DrawingURL* property to load a drawing from a URL.

To load the drawing into a **C# program**, you can also use the *LoadWidget* method of the *GlgObject* class or the *LoadWidget* method of the *GlgControl*.

Loading a Drawing into an ActiveX Control

To use a drawing with an ActiveX control, it must contain a viewport named *$Widget*. This viewport will be displayed when the drawing is loaded.

The ActiveX control has the *DrawingFile* property that specifies a drawing file. Setting this parameter loads the drawing. The control's *LoadWidgetExt* method can also be used to load a drawing.

The GLG ActiveX control also has the *DrawingURL* property for loading a drawing file from a URL.

Refer to the *Using the ActiveX Control* chapter of the *GLG Programming Guide* for more details.

Loading a Drawing into a JavaScript Program Deployed on a Web Page

To use a drawing in a JavaScript program, it must contain a viewport named *$Widget*. This viewport will be displayed when the drawing is loaded.

The drawing is loaded using the *LoadWidgetFromURL* method of the GLG JavaScript API. The DEMOS_HTML5 and examples_html5 directories contain source code examples of using GLG drawings in the HTML5 JavaScript environment.

Supplying Data for Animation from a Program

When the drawing is loaded into a C/C++, Java, C#/.NET or JavaScript program, or an ActiveX Control, the GLG resource mechanism is used to access resources of the drawing and to supply new resource values for animation, in the same way you did when prototyping the drawing in the Builder.

All GLG containers for different programming environments provide **SetResource** methods for setting named resources in the drawing. The resources may be of the *D* (double), *S* (string) or *G* (XYZ or RGB triplet) type. The names and syntax of methods slightly differ depending on the programming environment used (C/C++, Java, C#/.NET, JavaScript or ActiveX), but they are used the same way in all environments. The **Update** method is used to redraw changes on the screen after setting the new resource values.

For example, the following fragment from the GLG Animation examples "blinks" the object 3 times by alternating it's color:

```
// Change the ColorIndex of the color dynamics attached to
// the object's FillColor attribute from 0 to 1 and back
// to blink.
for( j=0; j<6; ++j )
{
   GlgSetDResource( viewport, "CatchMe/ColorIndex", (j % 2) ? 1. : 0.
            );
   GlgUpdate( viewport );
   GlgSleep( 100 ); // Delay for 0.1 sec
}
```

In this example, the *GlgSetDResource* method is used to set the new value of the *"CatchMe/ColorIndex"* resource, and the *GlgUpdate* method is used to update the graphics making the changes visible.

Refer to the *GLG Programming Guide* for more details.

Source Code Examples

The GLG *examples_c_cpp* directory contains C and C++ source code for a simple GLG animation examples that shows how to load and display a drawing, animate it with data and handle user interaction with the drawing.

DEMOS directory contains source code for more elaborate C/C++ demos: an aircombat simulation, a GIS map demo, a diagram editor demo and others.

DEMOS_JAVA directory contains the source code of Java demos.

DEMOS_C# directory contains the source code of the C#/.NET demos.

DEMOS_HTML5 directory contains the source code of the HTML5 and JavaScript demos.

examples_c_cpp directory contains C and C++ source code examples.

examples_java directory contains Java source code example*s*.

On Windows, additional examples listed below are also available.

examples_csharp.NET contains C# examples that use the GLG C# Class Library.

examples_csharp_ocx and ***examples_vbnet_ocx*** directories contain .NET examples that use the GLG ActiveX Control.

examples_ASP.NET and ***examples_jsp*** directories contain examples of the GLG Graphics Server for the ASP.NET and JSP. These demos are also available on-line at the following link:
http://www.genlogic.com/ajax_demos.html

5. Creating the Animation Example's Drawing

This section explains the design of the drawing used for the GLG Animation example in the GLG *examples* directory. To learn how the drawing is built, this section lists step-by-step instructions for building the drawing from scratch. It is assumed here that you're already familiar with the basic GLG Builder navigation described at the beginning of this tutorial.

The drawing we'll be building here will display a bouncing ball with an attached X and Y move dynamics that is used to animate the ball's movement. The ball also has a *Fill Color* dynamics that is used to flash colors when the ball is hit with the mouse.

Creating a Drawing's Viewport

To create this or any other GLG drawing that will be used with a program, start with a new widget (*File, New Widget*): this will create a new viewport and bring the editing focus into it.

Use the *Properties*  toolbar button to bring up the viewport's *Properties* dialog and click on the *Screen Properties* button. Set *Stretch XY*=NO to preserve X/Y ratio of the drawing: we are going to create a circle object and this will prevent the circle from being "squished" when the drawing is resized. Set *PushIn*=YES: this makes the default [-1000;+1000] coordinate extent to always appear in the viewport's window regardless of its X/Y ratio.

Creating a Circle Object

Create a small circle in the center of the drawing by clicking on *Filled Circle*  from the *Object Palette* and selecting two points in the drawing area: the first point will define the circle's center and the second will define its size. Position the circle in the center of the drawing by selecting the drawing's center as the circle's center point, or moving the circle after it has been created.

Use the *Properties*  toolbar button to bring up the *Properties* dialog and name the circle *CatchMe*. Change the circle's *HasResources* flag to **YES** to define the resource hierarchy: all circle's resources will be referenced by using a *"CatchMe/"* path prefix.

Adding Color Dynamics

Click on the ellipsis button  next the *Fill Color* attribute to activate the *Attribute* dialog, then select the *Add Dynamics* button at the bottom of the dialog. Select *List* from the list of color dynamics choices. This will attach the list color dynamics to the circle's *Fill Color* attribute and will bring up the *Edit Dynamics* dialog showing the dynamics we just attached.

Type the *ColorIndex* name in the right-most field for the dynamics' *Value Index* parameter. This resource name will be used to change the color displayed. The value of 0 will display the first color, the value of 1 will display the second color, and so on.

Select the ellipsis button  next to the *List of Values* to display the list of colors. Select the first item in the list and define it's color by selecting a red color from the palette, then select the second item and select a yellow color.

Try changing the value of the *Value Index* parameter from 0 to 1: the circle's colors should change from red (0) to yellow (1). Restore the index value to 0 and close the dialog.

Adding the Move Dynamics

Make sure the circle object is still selected and click on the *Add Dynamics* toolbar button to bring up the *Add Dynamics* dialog. Set the transformation type to *MoveBy* at the top of the dialog, set *Move Direction=X* and hit *Apply*. This attaches a *MoveX* transformation to the circle and brings up the *Edit Dynamics* dialog showing its attributes.

Set *XDistance*=0 and name the attribute *XValue* in the *Name* field to the right of it - this name will be used to set the set the X movement amount when animating the object. Set relative *Factor*=1, so that the effective movement value (defined as the multiplication of the *Factor* and *XDistance*) is completely controlled by the *XDistance* attribute.

Click on the *Add Dynamics* toolbar button again, change *Move Direction* to *X* and hit *Apply*. This attaches a *MoveY* transformation to the circle. Set *YDistance*=0 and name the attribute *YValue*. Set *Factor*=1.

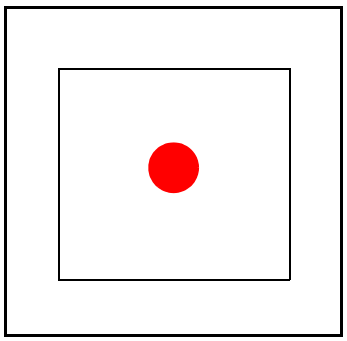
The *MoveX* and *MoveY* transformations will be used to animate circle's movements by changing their *XValue* and *YValue* resources.

Close the *Edit Dynamics* dialog.

Creating an Area Polygon

Unselect the circle by pressing the *ESC* key and select *Options, SnapTo, To Grid* from the main menu to make it easier to create a rectangular polygon.

Use the *Closed Polygon* *Object Palette* icon and select 4 points in the drawing to create a rectangular polygon as shown in the following picture:



Bring up the *Properties* dialog for the polygon, name it *Area* and set its *HasResources*=YES. *Shift*-click on the polygon's lower left control point to bring its *Control Point Editing* dialog and name the point *LLPoint*. *Shift*-click on the polygon's upper right control point and name it *URPoint*. These names will be used by the program to access the points, to query the extent of the Area polygon. The program will then use the extent to animate the circle, moving it within the Area boundaries.

Close the dialogs.

Adding Buttons

Click on the *Custom Objects*  icon in the *Object Palette* to bring up the *Custom Objects* dialog and select a *Button* object from the dialog. This will paste a native button object into the drawing. Place two buttons at the bottom of the drawing and name them *Slower* and *Faster*. These buttons will be used to change the moving ball's speed when the user clicks on a button. Reset the drawing by clicking the *Reset*  toolbar button to force the buttons to display the new names.

Click the *Resources*  toolbar button to bring up the *Resource Browser*, double-click on the *Faster* resource, then select the *TooltipString* resource and set it's value to *Increase Speed*. This value will be used to display the *Faster* button's tooltip in the run mode.

Double click on the *".."* item in the *Resource Browser* to return to the top level, double-click on the *Slower* resource, and set it's *TooltipString* to *Decrease Speed*.

Testing the Tooltips

Reset the drawing by clicking the *Reset*  toolbar, and select the *Start*  toolbar button to prototype the drawing. Enter an empty animation script and hit *OK* to start prototyping. Position the mouse over one button then another and wait to see the tooltips coming up. Stop the prototyping by clicking on the *Stop*  toolbar button.

Using the Drawing

We now finished creating the drawing. Rename the old *animation.g* drawing in the GLG's *examples* directory to *animation_bak.g* and save the drawing as *animation.g* in the *examples* directory by using *File, Save As* from the main menu. You should now be able to run the example using the new drawing.

If you get errors running the example with the new drawing, check the spelling, letter capitalization and resource paths of resource names reported in the error messages. Use the *Resource Browser* to check the names in the Builder: the names and resource paths should appear exactly like they are reported in the error messages for the resources that aren't found.

6. Using GLG Real-Time Charts

The *Real-Time Charts* widget set contains a collection of high-performance charts optimized for real-time rendering of large data sets. They use an integrated *Chart* object that enables fast rendering with support for integrated zooming and scrolling, chart tooltips and cursor feedback.

Editing Chart Properties in the Builder

The run-time behavior and appearance of a chart widget is controlled by the properties of the *Chart* object contained in its drawing. While these properties can be set in a program at run-time, the chart can be customized using the Graphics Builder, saving a modified drawing into a file.

Loading Charts into the Builder

The *Real-Time Charts* palette contains various real-time chart widgets. To load a chart into the Builder, select *Palette, Real Time Charts* from the main menu to display the palette, then *Ctrl*-click on a chart in the palette to load it. Press *OK* to discard the current drawing.

When a chart is loaded using *Ctrl*-click, the drawing will also contain a sample animation command that can be used to prototype the chart in the Run mode.

To add a chart to an existing drawing without replacing the drawing's content, click on the chart in the palette without holding down the *Ctrl* key. Refer to the *Loading Widget Drawings into the Builder* section on page 45 for more information.

Editing Chart Properties

A chart widget consists of a viewport named *$Widget* which contains objects used to render the widget: the *Chart* object, a text object named *Title* and an optional *Legend* object for multi-line charts.

The *Chart* object defines the appearance and run-time behavior of the chart; its properties are used to edit the chart. The *Public Properties* dialog provides a convenient way to access the chart's properties. Click on the *$Widget* viewport to select it, then use either the *Public Properties* toolbar button  , the *Public Properties* popup menu option, or the *Object, Public Properties* menu option to bring the *Public Properties* dialog.

The dialog will have the *Edit Chart* button for accessing properties of the *Chart* object inside the widget. The *Edit Legend* button will also be present for charts that have a legend. The dialog will also have the *Background* button for accessing rendering properties of the viewport, and the *Offsets* button for changing offsets that control layout of the *Chart* and *Legend* objects inside the widget.

Click on the *Edit Chart button* to bring the *Selected Object Properties* dialog showing the Chart's properties. Use the dialog's menus to edit the Chart's properties (see the *Editing Object Properties* section on page 14 for details).

Refer to the *Chart* chapter on page 128 of the *GLG User's Guide and Builder Reference Manual* for a description of the *Chart* object's attributes. Attributes of the chart's *Plot*, *Axis* and *Legend* objects are described on page 138, page 149 and page 159 of that manual.

Click on the *Back* button to return to the *Public Properties* dialog.

Alternatively, the *Chart* and other objects inside the widgets can be edited directly. Make sure the *$Widget* viewport is still selected, then click on the *Hierarchy Down* button to go "down" into it. Click on the chart's data area to select the *Chart* object: it's name (*Chart*) and type (*CHART*) will be shown in the status area at the bottom of the Builder. The chart's control points can be used to change the chart's layout (see the *Selecting an Object and Changing Object Geometry* section on page 12 for details).

Use the *Properties* toolbar button to bring up the *Selected Object Properties* dialog showing the chart's properties. Alternatively, you can use the right mouse button and select *Properties* from a popup menu, or *Object*, *Properties* from the main menu. The *Selected Object Properties* dialog can be used to edit chart's properties and is the same dialog that was accessed using the *Edit Chart* button in the previous example. The *Title* and *Legend* objects can also be selected and edited.

Use the *Hierarchy Up* button to get back to the top level.

Editing Chart Plots and Axes

Use the *Edit Chart* button is the *Public Properties* dialog to access chart's properties as described above. To access properties of the chart's plots, use the *Edit Plots* button. It will display a dialog with a list of plots; selecting a plot will display its properties in the *Selected Object Properties* dialog. Use the *Edit Line* button to get access to the plot's rendering attributes, such as *EdgeColor*, *LineWidth* and *LineType*.

Use the *Back* button at the bottom of the dialog to get back to the chart's properties.

Similarly, the *Exit Y Axis* button provides an access to the properties of the chart's Y axes and activates a list dialog for selecting a Y axis.

Properties for Supplying Chart Data

Each plot in a chart has three properties for supplying data:

- *Value Entry Point* for supplying Y values
- *Time Entry Point* for supplying time stamps for real-time Strip Charts, or X values for XY Scatter Plots
- *Valid Entry Point* for supplying data points' Valid flags

The entry points are present in the Plot object's properties dialog; they can also be accessed via resources in a program to supply chart data at run time. Refer to the *Plot* chapter on page 138 of the *GLG User's Guide and Builder Reference Manual* for a detailed description of a plot's entry points.

Editing Charts Using Resources

The chart widget can also be edited using resources. Resources are also used in a program to configure the chart and supply its data at run time.

In the Builder, resources can be accessed via the Resource Browser, which can be used to traverse the chart's resource hierarchy and modify the chart's properties via the resource mechanism.

To browse chart's resources, go to the top level and click on the chart widget's *$Widget* viewport to select it (use the *Hierarchy Up* button to get back to the top level if necessary). Click on the *Resources* toolbar button to open the *Resource Browser*; it will display a list of the chart's resources.

Resources are organized hierarchically: a list of chart resources contains *Chart* and *LegendObject* resources, which are composite objects containing other resources inside. Such composite resources are annotated with the >> suffix added to their names. Double-clicking on such composite resources shows resources inside them. For example, double-clicking on *Chart*, *Plots*, *Plot#0* shows resources of the chart's first plot. Selecting the *EdgeColor* resource activated a dialog with a color palette to edit the resource's value.

The Resource Browser provides a convenient interface for determining proper resource names that will be used in a program to set the chart's properties and supply data at run time. For example, the *ValueEntryPoint*, *TimeEntryPoint* and *ValidEntryPoint* resources of each plot are used to supply data at run time. When each of these resources is selected in the Resource Browser, the complete resource path is shown in the *Selection* field at the top of the Resource Browser dialog. This resource path is used in a program to access the corresponding resource at run time.

Double-click on the ".." entry to return to the previous level of the resource hierarchy, one level at a time. Alternatively, click on the chart in the drawing to get back to the top level.

Refer to the *Using Resources* chapter on page 323 of the *GLG User's Guide and Builder Reference Manual* for more information on using resources.

Prototyping the Chart's Run-Time Behavior

To animate the chart with simulated data, click on the *Start* toolbar button, then press *OK* to accept the default animation command. It will start the Run Mode and animate the chart with data generated by the default animation command supplied in the chart's drawing (the chart widget must be loaded by *Ctrl*-clicking in the Real-Time Charts palette in order for the animation command to be present in the drawing).

The Run Mode toolbar contains controls that control the update speed and display performance data. The Pause button pauses data updates, which makes it easier to test chart tooltips. Moving the mouse over the chart activates a cross-hair cursor lines that follow the mouse. A tooltip for the closest data point is displayed when the mouse hovers over a plot. An axis tooltip is displayed when the mouse hovers over a chart axis.

Use the *Stop* button to return to the editing mode.

7. Using Legacy GLG 2D and 3D Graph Widgets

The *2D Graphs* and *3D Graphs* widget sets contain a variety of graph widgets that compliment the *Real-Time Charts* widget set. While the *Real-Time Chart* widgets provide superior performance and advanced interaction capabilities, the 2D and 3D graphs may provide more elaborate rendering as well as additional graph types, such as packed, stacked and polar charts.

The below is a brief overview of 2D and 3D Graph widget usage. See the *2D and 3D Graphs* chapter on page 38 of the GLG Widgets Reference Manual for more information.

Loading Graph Widgets into the Builder

To load a graph widget into the Builder, select *Palettes, 2D Graphs* or *Palettes, 3D Graphs* from the main menu, then *Ctrl*-click on a graph in the palette to load it. Press *OK* to discard the current drawing.

When a graph widget is loaded using *Ctrl*-click, the drawing will also contain a sample animation command that can be used to prototype the graph in the Run mode.

To add a graph to an existing drawing without replacing the drawing's content, click on the graph in the palette without holding down the *Ctrl* key. Refer to the *Loading Widget Drawings into the Builder* section on page 45 for more information.

Two sample graphs, *bar1.g* and *line1.g,* are provided in the *Custom Objects* palette (the *Custom Object*  icon in the *Object Palette*).

Common Graph Resources

While the graphs have a lot of resources for customization, most of them have a common resource set that controls the graph:

Data Supply Resources

DataGroup/EntryPoint
Entry point for supplying graph data

XLabelGroup/EntryPoint
Entry point for supplying graph labels

Title resources:

Title/String
Graph's title

XAxisLabel/String
X axis title

YAxisLabel/String
Y axis title

DataGroup Resources:

DataGroup/Factor
Number of graph datasamples

DataGroup/ScrollType

Graph's scroll type: 0 - wrap, 1 - scroll

DataGroup/Inversed

Graph's scroll direction

DataGroup/DataSample/High

Graph's high range

DataGroup/DataSample/Low

Graph's low range Range resources

Axis Label Resources

YLabelGroup/YLabel/Format

C format for label display YLabels Resources

YLabelGroup/Factor

Number of labels and major ticks for Y axis

YLabelGroup/MinorFactor

Number of Y minor ticks X Label Resources

XLabelGroup/Factor

Number of labels and major ticks for X axis

XLabelGroup/MinorFactor

Number of X minor ticks

Refer to the *GLG Widgets Reference Manual* for a complete resource list.

Graphs with Multiple Data Groups

You can use graphs other than the basic *bar1.g* and *line1.g* to display more than one data set or display data in 3D.

Graph Widget Animation Examples

Examples in the *Supplying Data for Animation from a Program* section on page 75 lists some typical resources that are used to animate the GLG graphs.

Additional C/C++ source code examples may be found in the *examples_c_cpp* directory of the GLG installation. Java examples are located in the *examples_java* directory, and .NET examples are located in the *examples_csharp.NET, examples_csharp_ocx* and *examples_vbnet_ocx* directories. The *examples_html5* directory contains HTML5 and JavaScript examples.

8. Using Containers and SubDrawing Objects

Containers and *SubDrawings* represent two types of wrapper objects that may be created around a group of objects. They have a single control point that defines their position and are often used as wrappers around complex objects, such as nodes or icons, making it easier to position them. *Containers* and *SubDrawings* are subtypes of the *Reference* object.

All drawings described in the tutorial may be found in the *<glg_dir>/tutorial* directory. These drawings may be used to verify the final result of the steps described in the following chapters.

Container Object

A *Container* object is the simplest wrapper for positioning complex objects. Let's try to create one and see how it is used.

Creating a Template

Start with a new drawing by selecting *File, New Widget* from the main menu. Using the *Filled Polygon*  button, create a filled triangle centered around the center of the drawing (the center of the drawing is marked with the axis icon). Bring the polygon's *Properties* dialog, name it *Polygon* using the *Name* field. The polygon's *HasResources*=NO.

Let's add rotation dynamics to the polygon. Click on *Add Dynamics* button and select *Rotate* for transformation type in the *Add Dynamics* dialog. Set *Rotation Axis* to *Z* to rotate in the XY plane of the drawing. Set *Center* parameter to "0 0 0", which coincides with the center of the polygon. At this point, it doesn't matter what the Angle parameter is set to.

Click on *Apply* to attach the transformation; the *Edit Dynamics List* dialog will come up. Set the *Factor* parameter to 1. Set *ZAngle* parameter to 0 and name this parameter *Angle*. Click on *OK* to close the dialog.

Make sure not to move the polygon with the mouse at this point, since it will distort its position relatively to the rotation center. Later on in this chapter, we'll talk about preserving the position of rotation center relative to the object, so that it is not affected by the object move.

Create a fixed text object with a text *label* and position it below the triangle. Bring its *Properties* dialog and name the text object *Label*. Click on the ellipsis button next to the *TextString* attribute to bring its *Attribute* dialog, and name the attribute *LabelString* using the *Name* field.

Click on the *Group*  button and create a group object containing both triangle and label objects. Bring its *Properties* dialog and name the group *IconGroup*. Notice its *HasResources=NO* setting.

You can think of this group object as a graphical icon composed of a triangle and a label. The triangle may represent some live object (for example, an airplane) and the label may be used to display some textual information next to it, such as a flight number. The triangle may be rotated around its center to annotate airplane's current direction. When such icon is used in a program, it needs to be positioned at the exact latitude/longitude coordinates of the airplane it represents. It

would be convenient to be able to position the icon by setting coordinates of a single control point, which could be accessed as a resource from the program. A container object does exactly that - it allows to create a "wrapper" around a group of objects, with a single point to position it.

Creating a Container Object

While the group object is selected, create a container object by clicking on the *Container* button in the *Object Palette*. Define a position of the container's control point by clicking in the center of the triangle - the control point will be attached at that place.

Bring up the container's *Properties* dialog and notice that its *ReferenceType* is set to *Container*. Name the container *Icon* and set its *HasResources=YES*. Notice the *FixedSize* property which, if set to YES, may be used to create icons of a fixed size. The fixed size containers do not resize when the drawing is zoomed or resized, even if the drawing itself is resizable. The only way to change the size of a fixed size container is by editing or scaling its template.

Shift+click on the control point to bring its *Control Point* dialog and name the point *Position*. Try changing the coordinates of this point to "200 0 0". The object will move, with its control point (the center of the triangle) placed at the location (200,0,0). The *Position* resource may be accessed from a program to position the icon.

With the *Icon* object selected, select *Object, Resources* from the main menu (or *Resources* toolbar button) to bring the *Resource Browser*. It will show the following resources of the *Icon*:

- *Position*, defines the icon's position
- *LabelString*, defines the text string of the label displayed below the icon
- *Angle*, defines the triangle's rotation angle
- *Polygon*, provides an access to the triangle's attributes, such as *FillColor*, etc.
- *Label*, provides an access to the triangle's attributes, such as *FillColor*, etc.
- *Template* and *Origin* resources, which will be explained later.

Notice that even though *LabelString* and *Angle* resources are parameters of the group's text and polygon object respectively, they are visible at the *Icon* level, since *HasResources*=NO for both the text, the polygon and the group. It makes the resources visible at the container's level, making it possible to access them as *Icon/LabelString* and *Icon/Angle*. This is convenient, as we can access icon's attributes without knowing its exact object hierarchy.

Set the *LabelString* resource to *Plane 1100*, set the *Position* resource to "400 0 0", and change *Polygon/FillColor* resource to red using a color palette. The icon will move to a new position and will be displayed in the new color. The icon's label will display a new text as well. When the drawing is deployed in a program, the same resource settings can also can be done programmatically using the GLG API to update the icon with real-time data.

Set the *Angle* resource to 30. Notice that although the icon has moved, the polygon is still rotated around its center, meaning that the relative position of the rotation center is preserved when the object moves. In fact, the container object preserves the relative position of rotation center by moving both the object and the rotation center, so that their position relatively to each other does not change. The same is true for the scale center if a scale transformation is used.

The *Resource Browser* also displays the container's *Origin* and *Template* resources. The *Origin* defines the attachment position of the container's control point.

The group object used to create the container is called a template, and is displayed as a *Template* resource in the *Resource Browser*. For the container object, the *Template* and *IconGroup* resources displayed in the *Resource Browser* refer to the same object: the group containing the triangle and label objects. *Template* is a default resource name which may be used to access the container's template when it is unnamed. In our case, since we set template's *HasResources*=NO to make its resources visible at the level of the container object, both the *Template* and *IconGroup* show no resources inside them in a resource browser.

Creating Container Instances

Let's try to make multiple instances of the icon. Create a copy of the *Icon* object by selecting *Edit*, *Full Clone* (or using the *Ctrl+L* key). The copy will be selected. Position it in the drawing using the mouse, then bring its *Properties* dialog and name it *Icon2*. Click on the *Resources* toolbar button to display *Icon2*'s resources, and set the resource values as follows:

Position = 400 400 0
LabelString = Plane 1200
Angle = 0
Polygon/FillColor = yellow color from the color palette.

The copy will reflect the new resource values. The changes are persistent: if you save the drawing and load it back, or reset it using the *Reset* toolbar button, the resource setting will be preserved for both the *Icon* and *Icon2* objects.

Save the drawing as *container.g*.

Editing Container's Template

While resources provide a way to access the template's attributes, the container's template is still accessible for direct editing, such as changing triangle's geometry or changing the position of the label object. Let's try to edit the *Icon2*'s template.

Select the Icon2 object with the mouse and click on the *Hierarchy Down*  button to go "down" into the container. The container's template will appear in the center of the drawing. Click on the triangle: it will select the group object which holds the triable and the label. Click on the *Hierarchy Down*  button again to go down into the group. Select the triangle and move one of its control points to change the triangle's shape. Click on the *Hierarchy Up*  button twice to get back to the drawing level, the *Icon2* object will be selected again.

Notice that only the *Icon2* instance changed, and the *Icon* object was not affected by the change. **That is because each instance of container objects keeps its own, unique copy of the template, allowing each instance to modify it.** That is not true for the *SubDrawing* object discussed below, which shares the template between the instances.

When traversing down into the container's template, the *Origin* attribute may also be edited directly. The *Origin* is displayed as a round marker, which can be selected and moved with the mouse. The template is drawn above the *Origin*, so in our case it will obscure it. To make the Origin marker visible, select the template and reorder it using *Arrange, Reorder, Move To Back*. The round marker will become visible in the center of the drawing. If you move it to one of the triangle's corners, the container's control point will now be attached at that corner

when you go back up. The attachment point of the container's control point may also be changed by moving the control point with the mouse while simultaneously pressing both the *Control* and *Shift* keys.

SubDrawing Object

A *SubDrawing* object is a more complex wrapper object used to replicate instances of a shared template in a drawing or in multiple drawings. The template is shared among the instances and can be changed in one place for all of them. There are three types of subdrawing objects that differ in the way they store their template: included subdrawings, file subdrawings and palette subdrawings.

Included SubDrawing

Included SubDrawings store the template in the drawing and replicate instances of the template in the drawing. Since the template is stored in the drawing, changing the template will affect only subdrawings in the current drawing.

Reusing a Template from the Previous Example

Let's create an included subdrawing using the template from the previously saved *container.g* drawing. Load the drawing using *File*, *Open*, select the *$Widget* viewport and traverse down into it using the *Hierarchy Down* button. Select the *Icon* container object and traverse down to its template as well. Click on the triangular polygon to select the group object containing both the polygon and the label. Click on the *Copy* toolbar button to copy the group object.

Create a new drawing using *File, New Widget*. Click on the *Paste* toolbar button to paste the group object into the drawing: it will be placed in the center of the drawing, the same way as in the original drawing. We are going to use it as a template for a subdrawing object we are about to create. Bring its *Properties* dialog and set its *HasResources*=YES, so that all template resources are visible at the template's level.

Creating a SubDrawing

To create an included subdrawing with the *IconGroup* object as its template, make sure the *IconGroup* is selected and click on the *SubDrawing From Object* button in the *Object Palette*. Click in the center of the triangle to select an attachment point (if you want to position it exactly in the center of the drawing, click on the *Point Value* button and enter *"0 0 0"*). A text entry dialog will come up prompting you to enter *ObjectPath* and *OriginPath*. Press *OK* to use defaults for now.

Bring the *Properties* dialog of the created subdrawing object. Name it *Icon* and set its *HasResources*=YES. The following properties are present in the subdrawing object's Properties dialog:

- *ReferenceType=SubDrawing* (as opposed to *Container*)
- *Source=Template*, meaning that a template is saved with the drawing
- *SourcePath*= 'blank' (defines a template path when *Source* is set to *File* or *Palette*)
- *ObjectPath*= 'blank' (defines an icon path inside a template with several icons)
- *CloneType= STRONG*, defines what resources, if any, are constrained among the instances. If *CloneType*=STRONG (default), resources that are *Global* will be constrained.

Click on the *Resources* toolbar button to display resources of the *Icon* object we just created. The resource browser shows a list of *Icon*'s resources, with the new *Source*, *SourcePath* and *ObjectPath* resources added to the list. The list also shows the *Template* resource, which is the *IconGroup* template object. There is also a new resource called *Instance*, which is a copy of the *IconGroup* template; it is used to render the subdrawing's instance. Resources of the *Instance* can be changed dynamically to animate each instance with corresponding data values.

For a subdrawing, the template is shared between all instances of the subdrawing object, and each instance creates a copy of the template and then uses it for rendering. **This is a major difference between a container and a subdrawing. Each instance of a container object has its own independent copy of the template, while all instances of a subdrawing share the template.**

Since both the template and its copy have the same name (*IconGroup* in our case), the *Template* and *Instance* default resource names provide a way to access both the template and its rendered copy via the resource mechanism. The *IconGroup* resource in the resource browser refers to the copy of the template, which is the same object as the *Instance* resource.

Only the template is saved with the drawing, while an instance is created dynamically during hierarchy setup by making a copy of the template. It means that when a drawing is deployed in an application, an *Instance* and its attributes are accessible only after hierarchy setup.

> The subdrawing creates a copy of its template at the hierarchy setup time. Before the hierarchy setup, the value of the Instance resource is *null*. After the hierarchy setup, the *Instance* resource is not *null*, and the *IconGroup* resource refers to the *Instance* object.

Double-click on the *Template* resource in the resource browser to see the list of its resources. Since *IconGroup* has *HasResources*=YES, the resources are visible at the template level. Double-click on the ".." entry in the resource list to go back to the resources of the *Icon* object, and double-click on the *Instance* resource: it contains the same set of resources as the template, but they refer to resources of the template copy, which is rendered on the screen. If you change the *Polygon/FillColor* resource of the *Instance*, it will be immediately reflected on the screen.

For subdrawings, the resources of both the template and its copy may need to be accessed, which is done via *Template* and *Instance* resources. Both *Template* and *Instance* have the same set of resources. Setting the template's *HasResources*=YES ensures that resources of both the *Template* and the *Instance* are accessible as resources of the corresponding object: *Template* or *Instance*. For example, the *Template/Polygon/FillColor* resource path may be used to access the fill color of the template polygon, and the instance's colors may be accessed using either *Instance/Polygon/FillColor* or *IconGroup/Polygon/FillColor* resource path.

Creating SubDrawing Instances and Editing the Template

Let's make a copy of the *Icon* object by selecting *Edit, Full Clone* (or using the *Ctrl+L* key). It will create a copy of the subdrawing object using the same template. The copy will be selected. Position it in the drawing using the mouse, then bring its *Properties* dialog and change its name to *Icon2*.

With the *Icon2* still selected, click on the *Hierarchy Down* button to get down to the subdrawing's template, the *IconGroup* object.

Click on the triangular polygon to select the group. Click on *Hierarchy Down* button again to go down into the group. Click on the triangle to select the *Polygon* object. Bring its Properties dialog, click on the ellipsis button next to its *FillColor* resource and change the color to blue using a color palette.

Go up the hierarchy by clicking on the *Hierarchy Up* button twice to return back to the drawing, the *Icon2* object is selected.

Notice that the color of the triangle for both *Icon* and *Icon2* objects has changed to blue, reflecting the change made to the template.

Let's try to change the template using resources. Bring the resource browser for the *Icon2* object and double-click on *Template*. Double-click on *Polygon*, then select *FillColor* and set it to green. Notice that the colors of subdrawings in the drawing did not change. That is because we changed the template, but the copies of it used for rendering by subdrawing instances did not change. Click on the *Reset* toolbar button: this will reinitialize the drawing, recreating template instances by copying the template with the new color, which is now shown on the screen. When we edited the template using the *Hierarchy Down* button, the drawing was reset automatically when we returned back up, without the need to use the Reset button. Traversing the hierarchy is the preferred way of editing the template.

Change the color back to blue and reset the drawing.

Let's try to change attributes of a cloned template instance. The *Instance* is not persistent and is recreated from the template every time the drawing is loaded or reset, so it can not be edited by traversing down to it. The only way to access it is using resources. Bring the resource browser for the *Icon2* object, double-click on *Instance*, then change *Polygon/FillColor* to red. The color of the triangle in the *Icon2* object has changed to red, while the triangle of the *Icon* object remained unchanged. This is different from changing the template's color: changing the color in the template changes all instances, while changing the color of a particular instance (*Icon2*) affects only that instance.

When subdrawings are used in an application to replicate some template, the application can change instance's attributes based on real-time data to implement icon dynamics. For example, if our template is used to represent an airplane, the *LabelString* and *Angle* attribute of each instance can be dynamically changed to indicate airplane's direction and flight number.

Now reset the drawing using the *Reset* button. The subdrawings will recreate the displayed instances by copying the template, and any changes made to the instances will be discarded.

While all changes made to the objects in the template are permanent and saved in the drawing, changes made to the instances are volatile and are lost when the drawing is saved and then reloaded, or when it is reset in the Builder.

Fixed Size SubDrawings

Same as for containers, the *FixedSize* attribute of a subdrawing object may be used to create icons of a fixed size. Fixed size subdrawings do not resize when the drawing is zoomed or resized, even if the drawing itself is resizable. However, a scale transformation can be attached to a subdrawing's template, and the scale factor of the transformation may be used to change the size of the subdrawing instances. This technique is used in the GLG AirTraffic Control and GLG GIS demos to change the size of the fixed-size airplane icons.

Rebinding Attributes of a SubDrawing

All instances of a subdrawing object in a drawing share the same template, causing them to be rendered using the same attribute values on initial appearance. There are cases, however, when it is desired that each subdrawing's instance in the drawing has its own settings of certain resources, and that these settings are saved in the drawing. It would mean that when the drawing is loaded, these resource values are preserved and not overwritten by the template.

It can be achieved by using attribute rebinding. To make an attribute value of a subdrawing instance persistent, that attribute has to be marked in a template by setting its *Global* flag value to *Bound*. Such marked attributes may have different settings for individual instances. The settings get saved in the drawing and restored when the drawing is loaded.

Let's see how to use this feature. Make sure the *Icon2* object is still selected, go down into it by clicking on the *Hierarchy Down* button. Select a group and go down into it as well. Select the triangle, bring its *Properties* dialog and click on the ellipsis  button next to its *FillColor* attribute.

In the *Attribute* dialog, change the *Local* setting to *Bound*. Bound attributes must be named, so name the attribute *IconColor* using the *Name* field.

Go up the hierarchy twice, so that the *Icon2* object is selected again. Bring the *Properties* dialog for the *Icon2* object, and notice that the *Edit Bindings* button is now enabled, providing an access to the binding array which holds values of rebound attributes.

Click on the button to see a list of bindings with the *IconColor* entry. Click on *IconColor* and change the color to yellow. The bindings array may also be accessed as the *Bindings* entry in the resource browser. Bring the resource browser for the subdrawing object and double-click on *Bindings*: you'll see the same list of bound resources.

Please note that the rebound color attribute might have been accessed via the resource browser using a resource path containing any combination of default attribute names and named resources (for example, *Icon2/Instance/IconColor* or *Icon1/IconGroup/Polygon/FillColor*) to achieve the same result, but editing bindings is more convenient, as it shows a single list of all bound attributes that can be assigned a custom value.

Reset the drawing using the *Reset*  button. Since polygon's *FillColor* is now *Bound*, its value is taken from the bindings array and is not discarded as a result of the reset operation. Likewise, this setting will be preserved when the drawing is saved and reloaded.

In addition to altering appearance of individual instances in the drawing, the bindings may also be used for assigning unique tags to each instance. Since all instances are cloned from the same template, defining a tag in a template would result in the same tag being assigned to each instance. If different instances need to be connected to different data sources via tags, rebound attributes may be used, and unique tags may be assigned to each rebound attribute in the *Bindings* array.

The bindings may also be used to constrain attributes of each instance to attributes of other objects in the drawing, as well as assign new names to rebound attributes.

Using Global Attributes

While some attribute settings, like airplane's icon *Angle* or *LabelString*, may need to be unique for each icon instance, other attributes, such as label color, may need to be constrained among all instances, so that the attribute's value can be changed in one place for all instances. To achieve this, the label's *TextColor* attribute must be defined as *Global*.

With the *Icon2* object still selected, go down into it, then select the group and go down into it as well. Click on the text object to select the *Label* object. Bring its *Properties* dialog and click on the ellipsis button next to its *TextColor*. In the *Attribute* dialog, change the *Local* setting to *Global*.

Go up the hierarchy twice, so that *Icon2* is selected again. Bring the resource browser, select *IconGroup/Label/TextColor* and change it to dark blue.

Notice that the label color for both icons has changed.

Attributes marked as *Global* are constrained among all instances, including the template. Resource values applied to *Global* attributes of one instance (or the template) are reflected in all other instances, as well as the template.

> Reset the drawing. The label color remains dark blue for both icons. It is not discarded, unlike the case when the attribute was *Local*.

To summarize the attribute settings topic for a subdrawing object, an attribute of a template can be marked as *Local*, *Global* or *Bound*. Each setting has the following meaning:

- ***Local* attributes are unique for each instance and their settings for individual instances are not saved in the drawing.**
- ***Global* attributes are constrained among all instances. Their settings affect all instances, including the template.**
- ***Bound* attributes are unique for each instance, and their settings for individual instances are saved with the drawing. The instance's Bindings property contains a list of all bound resources. Bound resources must be named.**

To proceed with the tutorial for subdrawing objects, we will continue working with the same drawing and modify it as necessary. Let's save the drawing as *included_subdrawing.g*.

Object Dynamics

When the subdrawing object was created, we used the default settings of the *ObjectPath* attribute. The *ObjectPath* attribute may be used to implement object dynamics, with a template containing several different objects to represent different states of an icon. The use of the *ObjectPath* attribute is discussed in the *Object Dynamics* chapter on page 95.

File SubDrawing

A file subdrawing is similar to an included subdrawing but keeps its template in a separate drawing file, making it possible to share the template between multiple drawings. Changes applied to the template drawing will be reflected in all drawings that use it.

The below steps describe how to create a file subdrawing.

Reusing a Template

We will reuse the template of the icon we have already created. Select the *Icon* object, go down into it, select *IconGroup* and copy it by using either the *Edit, Copy* menu option or the *Copy* toolbar button.

Select *File, New* menu option to create a new drawing. Please notice the use of *File, New* instead of *File, New Widget.* Since we are going to use the drawing as a template, we do not need the *$Widget* viewport.

Select *Edit, Paste* to paste the *IconGroup* object, then save the drawing as *icon.g*.

Creating a SubDrawing

Create a new drawing using the *File, New Widget* menu option. Click on the *SubDrawing From File* button to create a subdrawing object. Define subdrawing's position by clicking anywhere in the drawing. A *File Selection* dialog comes up asking to specify a subdrawing's filename. Enter *icon.g* and click on OK to use the saved template drawing. Another dialog comes up, asking to enter *ObjectPath*: enter *IconGroup* and press OK to use the *IconGroup* object from the template.

Bring the *Properties* dialog of the subdrawing object. Name it *Icon* and set its *HasResources*=YES. Notice that the file subdrawing's properties are very similar to those of the included subdrawing, with an exception that *Source=File*, instead of *Template*, and the *SourcePath* defines a filename of the subdrawing's template.

The value of *SourcePath* is an absolute path from the *File Selection* dialog, and it is a good idea to change it to a relative path, relative to the directory of the drawing. This makes the project's directory self-contained, with all connections between the files being preserved when the whole directory is moved to another location or another machine. When *SourcePath* specifies a relative path, the template drawing file (subdrawing) is searched for relative to the directory of the drawing first, and then relative to the current directory. The *GLG_PATH* environment variable may also be set to define a list of directories to search for subdrawing files.

Let's remove the directory path portion of the *SourcePath* property, leaving only the *icon.g* filename.

Click on the *Edit Bindings* button in the *Properties* dialog. Click on *IconColor* in the list of bindings to select it and change its color to, let's say, pink.

Shift+click on the subdrawing's control point and name it *Position*. In an application, the *Icon/Position* resource may be used to position the icon based on some values, for example latitude and longitude of an airplane represented by the icon.

Like the included subdrawings, the file subdrawings' instances are created dynamically during hierarchy setup. The attribute settings of bound attributes are saved in the drawing, and *Global* attributes are constrained among all instances of a subdrawing. Everything described above regarding attribute settings for included subdrawings also applies to file subdrawings. **The only difference between an included subdrawing and a file subdrawing is that for an included subdrawing a template is saved in the current drawing, while for a file subdrawing a template is stored in a separate file that can be shared among multiple drawings.**

The template of a file subdrawing is loaded at the drawing setup time. The template is then copied to create an instance of it used for rendering an instance of the subdrawing. Before the hierarchy setup, the value of both the *Template* and *Instance* resources of the file subdrawing object is *null*.

The subdrawing's *FixedSize* attribute may be used to specify the subdrawing's resize policy. If the *FixedSize* is set to *YES*, the subdrawing will maintain a constant size when the drawing is zoomed or resized, which may be used to create dynamic icons of a fixed size, as shown in the *AirTraffic Control* and *GIS* demos.

Accessing the Subdrawing's Template

In the Builder, the template drawing can be conveniently accessed by traversing down the subdrawing object. With the *Icon* object still selected, click on the *Hierarchy Down* button, to go down into the subdrawing's template. It will automatically attempt to load the subdrawing's template drawing, *icon.g*. The Builder will display the *File Selection* dialog, asking for confirmation to load this file. Press the *OK* button.

If desired, you can make changes to the template, or examine its objects and resources. For now, let's leave it the way it is and go up the hierarchy by clicking on the *Hierarchy Up* button. The *File Selection* dialog is displayed again, asking for confirmation to save *icon.g*. You may press *OK* to accept, or you may press *Cancel* to abort the save operation.

Since there were no changes, press *Cancel*, then press *OK* in the message dialog to confirm discarding any changes.

Save the drawing as *file_subdrawing.g*.

Subdrawing File Dynamics

The file dynamics may be used to change the subdrawings template, so that different graphics is displayed when the state of the object changes.

Let's create another template drawing, so that we can alternate between two templates to see how file dynamics work.

Load *icon.g* drawing using the *File, Open* menu option. Select *IconGroup* and go down into it using the *Hierarchy Down* button. Click on the triangle to select the *Polygon* object and delete it using the *Cut*  button. Create a filled circle in place of the polygon using the *Filled Circle*  button. Bring its *Properties* dialog, name it *Circle* and leave its *HasResources=NO*. Go up the hierarchy, so that the *IconGroup* group is selected again.

Save the drawing as *icon2.g*.

Let's implement a file dynamics to alternate icon representation between the *icon.g* and *icon2.g*. You may think of it as the icon having two states, one state is represented by *icon.g*, and the other state is represented by *icon2.g*. The file dynamics is implemented by attaching a *List* transformation to the subdrawing's *Source Path* property.

Load the *file_subdrawing.g* drawing. Select the *$Widget* viewport and go down into it. Select the *Icon* object and bring its *Properties* dialog.

In the Properties dialog, click on the ellipses button next to *Source Path* property to bring its *Attribute* dialog. Click on the *Add Dynamics* button at the bottom of the dialog and select *List*. Close the *Properties* dialog.

In the *Edit Dynamics* dialog for the list transformation, name the *Value Index* attribute *IconState*. The value of *IconState* is an index in a list of items, in this case a list of filenames to be used to draw the icon.

Click on the *List Of Values* button to display the list. In the *List Dynamics* dialog, select *Item0* and notice that its value in the *Attribute* dialog has been already set to *icon.g*.

Select *Item1* from the list and set its value to icon2.g in the *Value* field of the *Attribute* dialog.

Close all dialogs and bring the resource browser for the *Icon* object. Select the *IconState* resource and try alternating its value between 0 and 1. You'll see that when *IconState*=0, the *icon.g* file is used to represent the icon, and when *IconState*=1, the *icon2.g* file is used.

Save the drawing as *file_subdrawing.g*.

The file dynamics is similar to the image dynamics, used for image objects by applying a *List* transformation to the *Image File* attribute. An important difference is that the image dynamics switches static images, while the file dynamics switches dynamic graphical representations which may be further modified (by changing color, angle, label, etc.) based on real-time data.

Object Dynamics

As described in the previous chapter, file dynamics can be used to alternate the icon's appearance using different subdrawing files. While this is a powerful feature, its disadvantage is that a separate file should be supplied for each icon representation.

The Toolkit provides an alternative feature of *Object Dynamics*, where all possible representations of the icon are stored in a single drawing file.

Creating a Template with Multiple Icons

Let's create a single template file that contains two versions of the icon: a triangle from the *icon.g* file and a circle from the *icon2.g* file.

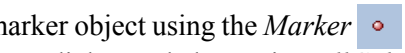
Open the *icon2.g* drawing using *File, Open*, select the *IconGroup* object and copy it by using either the *Edit, Copy* menu option or the *Copy*  toolbar button.

Open the *icon.g* file and paste previously copied object using either the *Edit, Paste* menu option or the *Paste*  toolbar button. Move the template with a circle so that it does not intersect the other template with a triangle.

Select the *IconGroup* object with a circle, bring its *Properties* dialog and rename it to *Icon2*. Click on the *Select Next*  toolbar button and click on the circle to select the *Circle* object (this is a shortcut to select individual objects in the group without going down into it).

The *Shift*+click on the control point in the center of the circle to brings its *Control Point* dialog and name the point *Anchor2*. It will be used to define the icon's attachment point in the center of the circle.

Press the *Escape* key to get out of the group traversal mode.

Select the other *IconGroup* object containing a triangle. Bring its *Properties* dialog and rename it to *Icon1*. Since the triangle does not have a control point in its center we could use to define the icon's anchor point, we need to create one. Create a marker object using the *Marker*  button and place it in the center of the triangle. Bring its *Properties* dialog and change its *FillColor* to red to make the marker more visible. *Shift*+click on the control point of the marker object and name it *Anchor1* in the *Control Point* dialog.

The *Anchor1* and *Anchor2* points will be used as origins (anchor points) for positioning the *Icon1* and *Icon2* templates in the drawing. We used two different methods to define an origin point. For *Icon1*, we created a marker object at the desired location and named its control point *Anchor1*. For *Icon2*, we used the circle's control point, naming it *Anchor2*, which will be visible as *Icon2/Anchor2* resource.

We now have a composite template drawing, or a palette, containing multiple representations of an icon: *Icon1* and *Icon2*. Save the drawing as *palette.g*. Let's now use it to create a subdrawing with object dynamics.

Creating a Subdrawing

Create a new drawing using *File, New Widget*. Create a subdrawing object by clicking on the *Subdrawing From File*  button. Define the subdrawing's position by clicking anywhere in the drawing. In the *File Selection* dialog, select *palette.g* as a subdrawing file.

Type *Icon1:Anchor1* in the *Text Entry* dialog for the *ObjectPath* and press OK. It directs the subdrawing to render the *Icon1* object of the template and anchor it using the *Anchor1* point.

Shift+click on the control point of the subdrawing and name it *Position* in the *Control Point* dialog. Click *OK* to close the dialog.

Bring the *Properties* dialog for the subdrawing object. Name it *Airplane* and set the *HasResources*=YES. The subdrawing's *Source* attribute is set to *File*, and its *Source Path* attribute points to the *palette.g* file. Change the *SourcePath* property so that it has only a filename (*palette.g*), without the directory path. Using a relative file name without the path makes it easier to move the drawing to other machines and directories.

The subdrawing object with the file dynamics we created earlier in this chapter (*file_subdrawing.g*) used several templates, each containing a single graphical representation of the icon. The new subdrawing object we just created, *Airplane*, uses a template subdrawing (*palette.g*) which contains multiple objects, or multiple representations of an icon.

Using ObjectPath for Object Dynamics

The *ObjectPath* property of the subdrawing object defines the name of an object in the template to use as an *Instance* for rendering the subdrawing. It also defines (after a colon) a name of the anchor point: this point of the template will be placed at the position of the subdrawing object's control point.

In our example, *Icon1* is the object name (the resource path in the *palette.g* file) of the template object to be used as a graphical representation for the *Airplane* subdrawing object. *Anchor1* is the resource path of the point to be used for "anchoring" the template object. This point will be placed (mapped) at the location of the subdrawing's *Position* point.

Now let's add object dynamics. Click on the ellipsis button next to the *ObjectPath* attribute to bring its *Attribute* dialog. Add *List* dynamics to the attribute by clicking on the *Add Dynamics* button and selecting the *List* transformation.

In the *Edit Dynamics* dialog of the list transformation, name the *ValueIndex* attribute *IconState*: its value will be used to select the list items. Click on the ellipsis button next to the *List Of Values* attribute to display the list. Select the *Item0* list item to bring its *Attribute* dialog and notice that its value has been already set to *Icon1:Anchor1*.

Select the *Item1* list entry and type *Icon2:Icon2/Anchor2* in the value field of the *Attribute* dialog. The *Icon2* object and the *Icon2/Anchor2* anchor point will be used to render the subdrawing when the list item is selected. Close all dialogs.

With the *Airplane* object still selected, display its resources using the resource browser. Select the *IconState* resource and alternate its value between 0 and 1: when *IconState*=0, the *Icon1* object is used to represent the icon, and when *IconState*=1, the *Icon2* object with a circle is displayed. The circular icon may be used when the airplane direction is not determined and there is no data to set its *Angle* attribute.

Object dynamics is used to alter the icon's appearance based on some value, indicating the icon's state. Object dynamics is implemented by attaching a list transformation to the *ObjectPath* attribute of a subdrawing object. The *SourcePath* attribute should point to a template drawing file containing multiple representations of the icon.

Unlike file dynamics, the object dynamics feature has an advantage of having a single drawing file (a palette file) that contains all representation of an icon, as opposed to having a separate file for each representation. The same palette file may be used across multiple drawings.

Save the drawing as *subdr_path.g*.

Palette SubDrawing

There is yet another type of a *SubDrawing* called a *Palette SubDrawing* which is a special case of an included subdrawing that stores its included template in a different way.

In cases when a subdrawing is used to visualize icons of different types, the template will contain a palette of icon shapes. An included subdrawing could be used to store the palette in the drawing in cases when it is not desirable to keep it in a separate file. However, the palette used as a template of an included subdrawing would not be visible at the top level of the drawing. A *Palette SubDrawing* provides an alternative mechanism for storing the palette in the drawing in a way which makes it easily accessible for editing even by users who are not familiar with the structure of the drawing.

Let's create a palette subdrawing. Load the *palette.g* drawing. Create a group object enclosing both *Icon1* and *Icon2*. Make a copy of the group by using either the *Edit, Copy* menu option or the *Copy*  toolbar button.

Create a new drawing using the *File, New Widget*. Go up the hierarchy, so that the *$Widget* viewport is selected.

Create a new viewport object by clicking on the *Viewport*  button and selecting two points in the drawing to define its area. Bring the viewport's *Properties* dialog, name it *$Palette* and set its *HasResources=YES.* Close the *Properties* dialog.

$Palette is a predefined name, like *$Drawing* or *$Widget*. It is used as a default palette name when Palette SubDrawing objects are created.

Use the *Hierarchy Down* button to go down into the *$Palette* viewport. Paste the previously copied template group with *Icon1* and *Icon2* objects using either the *Edit, Paste* menu option or the *Paste*  toolbar button. Explode the group by using the *Explode*  button, as the group is no longer needed.

Now we have a viewport named *$Palette* that contains two icon representations, *Icon1* and *Icon2*.

Go up the hierarchy. Select the *$Widget* viewport and go down into it.

Create a Palette SubDrawing object by using the *Object, Create, SubDrawing, SubDrawing From Palette* menu option.

Click anywhere in the drawing to define the subdrawing's position. In the *Text Entry* dialog, type *Icon1:Anchor1* as the value of the *ObjectPath* and press *OK*.

A "*Missing <$Palette> palette object*" error will be generated, and an "*Unresolved reference object*" text will be displayed in place of a newly created palette subdrawing. The error is generated because the *$Palette* object is outside of the scope of the current drawing. Ignore the error and click on the *Hierarchy Up* button to go to level where the palette object is visible. There will be no errors and the palette subdrawing will be displayed properly, with a triangular icon and a label.

Click on *Hierarchy Down* again to go down into the *$Widget* viewport, and notice that the are no errors: once the palette object is resolved, its location is remembered. Select the subdrawing and bring its *Properties* dialog. Name the object *Airplane* and set its *HasResources*=YES. Notice that its *Source=Palette* (as opposed to *File*) and the *SourcePath* is empty, causing the subdrawing to use the default palette name: *$Palette*.

> If the palette has a different name or located not at the top level of the drawing hierarchy, the SourcePath attribute defines the palette path. For example, for a palette named *MyPalette* located inside the *$Widget*

viewport, the *SourcePath* may be set to *$Widget/MyPalette*. The palette resource path is resolved at the drawing loading time, so it may be necessary to save and load the drawing back to get rid of the initial error when the palette subdrawing is just created.

The value of the *ObjectPath* attribute of the palette subdrawing is *Icon1:Anchor1*. It may be changed to *Icon2:Icon2/Anchor2* to display the second icon type with a circle. List dynamics may be attached to the attribute to switch icons based on the value of the list dynamic's *ValueIndex* attribute, as described in the *Object Dynamics* chapter above.

SubWindow Object

The *SubWindow* object is a special type of a subdrawing used to switch drawings displayed in the *SubWindow* object. The *SubWindow* has two control points that define an area in which the template drawing is displayed, and its template must be a viewport object.

The *SubWindow* object is created using the *SubWindow From File*  button using steps similar to the *SubDrawing* object. The only difference is that a subwindow uses two control points that define the subwindow's area, and its template must be a viewport. When prompted for a subwindow drawing, enter a path to a drawing file to be displayed in the subwindow. If the drawing file contains a viewport object named *$Widget, ObjectPath* may be left empty. If a different viewport name is used, it should be defined when prompted for *ObjectPath*.

To switch the drawing displayed in a subwindow, change its *SourcePath* attribute to point to a different drawing file.

The *SubWindow* may also be used as a subdrawing with two control points, which is useful for creating instances of interface objects such as buttons and menus. If a button template changes, instances of the button in all drawings will change as well. *Bindings* may be used to specify unique attribute values for each instance of the subwindow, such as a button label or a custom action ID. After a subwindow is created, its attributes may be changed to include the template instead of referencing a template in an external file.

Controlling Template Cache

The *EnableCache* attribute of the *SubDrawing* and *SubWindow* objects enables or disables a template cache for subwindows and subdrawings that use a template stored in an external file. If set to YES, the template is cached for reuse by subdrawings that use the same template file. Instead of loading the template multiple times, each subdrawing creates a copy of the cached template.

If set to NO, template caching is disabled and each subdrawing or subwindow loads its own copy of the template. It may be used to increase performance for subwindows that switch drawings: since only one copy of the drawing is loaded into the subwindow, it is more efficient to load it directly instead of loading it in the cache and then copying it to create a local copy of the template.

If *EnableCache* is set to NO, the *CloneType* attribute has no effect (attributes are not constrained). *EnableCache* has no effect for subdrawings and subwindows that use an included or a palette template.