GLG User's Guide and Builder Reference Manual

GLG Toolkit
Version 3.5

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

June 21, 2015
Software Release Version 3.5

Preface

GLG User's Manual and Builder Reference

This book provides information about creating and animating GLG drawings, using the GLG Graphics Builder and editing GLG objects. It contains the following chapters:

Table of Contents

Introduction to GLG
An overview of the GLG Toolkit and its components.

Structure of a GLG Drawing
A description of the internal structure of a GLG drawing. The material in this and the next chapter are important for anybody who wants to edit or create a GLG drawing.

GLG Objects
A description of each of the objects that make up a GLG drawing.

Integrated Features of the GLG Drawing
A description of the integrated features of the GLG drawing such as zooming and panning, object tooltips, custom selection events and commands, as well as other integrated features.

Input Objects
A description of the interaction handlers and input objects.

Using the GLG Graphics Builder
An introduction to the use of the GLG Graphics Builder. This program is used to create, edit, and test GLG drawings.

GLG Graphics Builder Menus
A reference for the GLG Graphics Builder menus and dialogs.

Index

This guide assumes that you are conversant with the basic concepts of computer graphics programming. For a comprehensive discussion of three dimensional computer graphics, we recommend *Computer Graphics: Principles and Practice*, Foley, Van Dam, Feiner, and Hughes. Second edition, 1990; Addison-Wesley, Reading MA.

Please note that although the illustrations in this document represent the UNIX version of the GLG Graphics Builder, the information it contains is equally relevant to Microsoft Windows users. The two versions present the same functionality in equivalent user interfaces, with minimal, cosmetic differences caused by the different platforms.

GLG User's Manual and Builder Reference

Table of Contents

Chapter 1

Introduction to GLG

1

The GLG Toolkit is used to create sophisticated real-time animated drawings. The design of a GLG drawing allows developers to edit drawings simply and quickly, without re-programming, and allows many simple tasks to be controlled within the drawing itself, without programming at all. For example, GLG allows a developer to create a graphical input widget to accept input from a user, and to link that input to some drawing feature, without programming a single line of code.

The ease with which you can create new and elaborate drawings and the speed with which you can adapt existing drawings to new uses make GLG ideal for custom data display projects. Also, the flexible structure of the product makes it adaptable to many different real-time display applications. This allows the application programmer to concentrate on the data collection and management aspects of such an application, instead of the display.

Overview

GLG consists of several separate components:

- The GLG Graphics Builder, used to create and edit GLG drawings.
- A set of GLG containers (GLG Bean, GLG Wrapper Widget, GLG ActiveX Control, etc.) for embedding GLG drawings into different programming environments.
- The set of functions (also called the Application Program Interface, or API) used to incorporate a GLG drawing into a user's application and update the drawing with real-time data.
- The extended set of functions (referred to as an Extended API) used to create GLG objects programmatically or perform complex manipulations on GLG objects.
- A library of ready-to-use GLG Widgets including graphs, meters, dials, avionics gauges, process control symbols and other widgets that can be used alone or incorporated into other drawings.
- Programming tools and utilities, including a data generator for prototyping animated drawings, a file format converter, and a tool for creating memory images of finished GLG drawings.

Central to each of these components is the GLG drawing. Broadly speaking, the GLG Graphics Builder is for creating and modifying these drawings, and the API is for including and controlling the drawings from a user's program. (There is also an "extended" API for creating and modifying a drawing from within a program.) The widget set is a library of GLG drawings, and the tools are aids for creating and editing these drawings. Rather than relying on the sophistication of the editor or the API, however, it is the organization and internal structure of these drawings that gives GLG its power. Because of this, it is important for you to be familiar with the general structure of a GLG drawing before trying to use the Builder or the API. Fortunately, though the GLG approach presents a rich set of possibilities to the user, the structure is not a complex one.

The organizing philosophy of the GLG system might be described as a process of relentless abstraction. Wherever possible, GLG uses the same data structure to describe similar objects. For example, a point in space uses three numbers to specify its position: its X, Y, and Z coordinates. Similarly, a color is described as a collection of three numbers: the red, green, and blue values. In GLG, these two kinds of data are represented using the same data structure. Similarly, a straight line segment and a complex polygon appear similar within GLG. After all, a line is simply a two-point polygon

In addition to keeping the size of a GLG application small, this object-oriented approach provides a tremendous amount of flexibility. An operation defined on a polygon, for example, may equally well be applied to only one of its vertex points, or to a neighboring object, or to the properties of any other object, including a color. On the other hand, with this flexibility comes some complications. The approach can create some unexpected side effects (linking colors to positions in space is just one example), and one of the important consequences is that there are usually several different ways to solve the same problem.

The following sections describe the components of the GLG Toolkit in greater depth.

The GLG Graphics Builder

The GLG Graphics Builder is used to create, modify, and test GLG drawings. It is the most sophisticated graphics animation editor available in its class, and lets you create elaborately structured animated drawings, and to test them with intricate animation data—generated or real. Since all aspects of the object's appearance and dynamics are encapsulated as data, most of the object dynamics and behavior may be defined and prototyped in the Builder without any programming. Elaborate **constraints** between objects' attributes may be created to define complex object behavior, and timer transformations may be used to animate objects without programming.

The Builder's data generator may be used to animate objects in the drawing with simulated data in the **prototype** mode, and the Builder's **resource browser** presents the user with resource interface to the drawing - the same interface the application will use at run time. The GIS Zooming Mode of the Builder assists in the setup and prototyping of the integrated **GIS mapping object**.

OpenGL or GDI (Native Windowing System) Renderer

OpenGL (Open Graphics Library) is a standard specification defining a cross-language cross-platform API for writing applications that produce 2D and 3D computer graphics. OpenGL is often used by 3D games and advanced graphical applications as an interface to the hardware accelerated graphics provided by modern graphics cards.

The OpenGL renderer is available for the GLG C/C++ applications on both Unix/Linux and Windows. On Windows, it is also available for the GLG ActiveX Control. Both the Graphics Builder and the GLG C/C++/ActiveX applications have a choice between the **OpenGL** renderer or a **native windowing system (GDI) renderer**, where GDI stands for Graphical Device Interface.

The OpenGL renderer uses hardware acceleration and enables such rendering features as antialiasing, true transparency and alpha-blending, native linear color gradients and hidden surface removal. The native windowing system renderer may be used as an alternative in cases when an application needs a greater consistency for rendering individual pixels regardless of the installed graphics cards. It may also be used as a fallback if the OpenGL renderer is not available.

The OpenGL renderer is used by the Toolkit in a transparent way, and the user does not need to know the low-level details of OpenGL graphics in order to benefit from the OpenGL's hardware acceleration and an extended set of rendering features. The same application executable may switch between using the OpenGL or native windowing renderer at run time, without any changes to the application code. The OpenGL renderer is supported in a cross-platform way on both Unix/Linux and Windows environments.

When the Toolkit is installed on Windows, two groups of icon shortcuts are provided for starting the demos and the Builder: one for the OpenGL rendering mode and another for the GDI mode. For Unix/Linux installations, symbolic links are provided for starting the demos and the Builder in either OpenGL or GDI rendering mode.

Java and .NET Note: Java and .NET use their own renderers that support anti-aliasing, transparency, alpha-blending and color gradients. The only feature of the OpenGL renderer not available in the Java and C#/.NET GLG applications is the hardware-accelerated hidden surface removal. For C#/.NET applications on Windows that need this feature, the GLG ActiveX Control provides an alternative to the GLG C# class library that supports OpenGL.

Enabling OpenGL renderer

The OpenGL renderer is enabled on per-viewport basis by setting viewport's *OpenGLHint* flag to *ON*. The flag may be set to several *ON* values with different rendering priorities described in the following section.

At run time, the OpenGL renderer may be enabled or disabled by using the *-glg-enable-opengl* or *-glg-disable-opengl* **command-line options**. Refer to the *Command-line Options* section on page 221 for more information.

The OpenGL renderer may also be enabled or disabled globally, by setting the *GLG_OPENGL_MODE* environment variable to *True* or *False*, or programmatically, by setting the value of the *GlgOpenGLMode* global configuration resource to 1 or 0. The global configuration resource takes precedent over the command-line options, while the command-line options take precedent over the environment variable settings.

OpenGL Versions, Compatibility and Core Profiles

By default, the OpenGL driver uses the Compatibility profile. If the graphics card supports OpenGL version 3.00 and above, a shader-based Core profile may be used for rendering. The Core profile uses VBO-based retained mode and does not provide immediate mode rendering. The retained rendering mode may provide performance improvements for drawings containing a large number of objects with static geometry. The retained mode may also significantly increase update speed of drawings with background images and text objects by storing cached textures on the graphics card. The retained mode is automatically activated when an OpenGL version 3.0 or higher is requested.

The *GlgOpenGLVersion* global configuration resource may be used to request a particular version of the OpenGL for the hardware-based version of the GLG OpenGL driver. For example, it may be set to a value of 330 to request OpenGL version 3.3. Alternatively, the GLG_OPENGL_VERSION environment variable and the *-glg-opengl-version* command-line option may be used to specify a desired OpenGL version.

The value of *GlgOpenGLVersion* is subject to the following thresholds:

- 100 (OpenGL version 1.0), uses glVertex()
- 110 (OpenGL version 1.1), uses vertex arrays
- 300 (OpenGL version 3.0), uses shaders, vertex arrays, as well as textures for text glyphs and images
- 330 (OpenGL version 3.3), uses shaders, VBOs, as well as textures for text glyphs and images.

If the requested OpenGL version is not supported by the graphics card, an error message is generated, and the driver is automatically downgraded to the highest supported OpenGL version.

Hardware and Software Renderers, OpenGL Priority

Graphics cards that provide hardware-accelerated OpenGL rendering have limitations on the maximum number of OpenGL windows an application can create (the exact number varies depending on a graphics card). To get around this limitation, GLG allows an application to combine both the hardware-accelerated and the software-based OpenGL renderer. A fast hardware-accelerated renderer may be used for main windows with lots of objects and fast update rates, while a slower software renderer may be used for icon buttons and other windows with small number of objects or with infrequent updates. As a result, applications with a large number of viewports can use OpenGL for all viewports without exceeding the limits of a graphics card.

At the design time, the application prioritizes viewports by setting the viewport's *OpenGLHint* attribute to one of several OpenGL priorities, from the highest (1) to the lowest (3). These priorities are used at runtime to determine the type of the OpenGL renderer (hardware or software) to use for each viewport. The hardware renderer is used for viewports with higher priorities, while the software renderer is used for viewports with lower priorities.

The *GlgOpenGLHardwareThreshold* and *GlgOpenGLThreshold* global configuration variables control the runtime mapping of the OpenGL priorities. All high-priority viewports with priority values less than or equal to *GlgOpenGLHardwareThreshold* will be rendered using the hardware OpenGL renderer. Low-priority viewports with priority values between *GlgOpenGLHardwareThreshold* and *GlgOpenGLThreshold* will be rendered using the software OpenGL renderer. Viewports with priority values equal 0 (disabled OpenGL) or greater than *GlgOpenGLThreshold* will use the GDI renderer.

The *-glg-opengl-hardware-threshold* and *-glg-opengl-threshold* command-line options, as well as *GLG_OPENGL_HARDWARE_THRESHOLD* and *GLG_OPENGL_THRESHOLD* environment variables may also be used to define the runtime mapping.

The global configuration resources and command-line options allow the user to change the runtime mapping on the fly. For example, if an application uses a small number of viewports, it can decide to use the hardware renderer for all its windows at runtime. If an application uses a large number of viewports, it can use software renderer for icon buttons and secondary viewports, while using faster hardware renderer for the main viewports. As a result, nice OpenGL anti-aliased rendering may be used for all application's viewports without exceeding the limits of a graphics card and its OpenGL driver.

OpenGL Setup and Diagnostics

To use hardware acceleration, a graphics card that supports OpenGL must be present and appropriate drivers must be installed. If the graphics card and/or drivers with the OpenGL support are not installed, a software OpenGL renderer will be used (always on Windows and Linux, and on other Unix systems - if available).

To check the status of the OpenGL renderer in the GlgBuilder, select the *Options / Display OpenGL Info* option from the main menu.

In both the Builder and the application, the ***-verbose*** command-line option may be used to display extended OpenGL diagnostic informations, including the version of the used hardware and software renderers. On Unix/Linux, the information will be printed to the terminal. On both Windows and Unix/Linux, the information will also be logged into the GLG log file named glg_error.log. The location of the log file is determined by the GLG_LOG_DIR and GLG_DIR environment variables as described in the *Error Processing* section on page 51 of the *GLG Programming Reference Manual*. The verbose mode can also be activated globally by setting the GLG_VERBOSE environment variable to *True*.

The *-glg-debug-opengl* command line option and the GLG_DEBUG_OPENGL environment variable may be used to generate extended information from the OpenGL driver. The output is logged into the *glg_error.log* file in the GLG installation directory, and is also printed in the command window on Unix/Linux.

OpenGL Libraries

The **hardware renderer on Unix/Linux** systems uses the *libGL* and *libGLU* libraries. These libraries are usually provided by the graphics card vendor.

On Linux, the *libGL* library from the *libgl1-mesa-glx* package provides support for hardware acceleration. The *libGL* library from the *libgl1-mesa-swx11* package supports only the software renderer.

For the **low-priority software renderer on Unix/Linux**, GLG uses the *libOSMesa* library (provided by the libosmesa6 package on Linux), as well as the *libGLU* library which provides some utility functions.

On Linux, the GLG editor uses the *libOSMesa* library provided by the GLG installation in the *glg/lib* directory for the software renderer. If the *libGLU* library is not provided by the hardware renderer, the editor also uses the *libtess_util* library from the *glg/lib* directory instead of the *libGLU* library. If a GLG application wants to use *libOSMesa* library from the *glg/lib* directory for the software renderer, it should include *glg/lib* in LDD_LOAD_PATH or copy the libraries to a place where they will be found by the linker.

An application can also set the *GlgOpenGLMesaLibPath* environment variable to point to the *libOSMesa* library to be used. In this case the *libtess_util* library will be searched in the directory where the *libOSMesa* library is located. If the *GLG_DIR* environment variable is set, the libraries will also be searched in the $*GLG_DIR/lib* directory.

On **Windows**, OpenGL uses Opengl32.dll and Glu32.dll libraries. These libraries are always present, but they support the hardware-accelerated renderer only if drivers for a graphics card with the OpenGL support are installed. Otherwise, only the software renderer is enabled. If hardware acceleration is enabled by the graphics card, the OpenGL libraries on Windows support both hardware and software-based rendering, with no additional libraries required.

On both Unix/Linux and Windows platforms, all OpenGL libraries are dynamically loaded at run time if they are available, so that the executable may be used even if the libraries are absent.

OpenGL on Linux Laptops with NVidia Optimus / Bumblebee

On Linux laptops with switchable graphics cards, the *optirun* command is used to run an application on the NVIDIA graphics card. The *optirun* utility preloads *libGL*, but not *libGLU* required by GLG. To run the Graphics Builder or a GLG application via *optirun*, *libGLU* has to be explicitly preloaded using the LD_PRELOAD environment variable. For example:

```
export LD_PRELOAD=/usr/lib/libGLU.so.1
optirun /usr/local/glg/bin/GlgBuilder
```

The Application Program Interface

Once a drawing is created, the GLG Toolkit offers a variety of ways to use that drawing in a program. Many of these methods are portable across window environments, allowing applications to be easily ported from X Windows to Microsoft Windows and back again.

Displaying a Drawing

The first step in using a GLG drawing from your application is to display the drawing on the screen. Again, depending on the application and graphical environment in which it will operate, you can choose one of several different display facilities:

GLG Wrapper Widget for C/C++ and X/Motif

You use the wrapper widget for applications that will operate in the X Windows environment. The GLG Toolkit supplies a version of this widget that uses Motif, and another that uses the Xt interface directly. You can use Motif if you want access to the Motif widgets, or if it's important for your application to use that graphical standard. The Xt version of the wrapper widget can also display and animate GLG drawings.

GLG Custom Control and GLG MFC Class for C/C# on Windows

This interface allows you to display and animate a GLG drawing in the Microsoft Windows environment.

Cross-Platform Generic API for C/C++

The functions of the GLG Generic API can be used to display and animate a GLG drawing in an entirely platform-independent way. A program using only functions from this API can work equally well in the Microsoft Windows and X Windows environments.

GLG Bean and GLG Class Library for Java

This 100% pure Java Class Library allows you to develop GLG applications using Java.

GLG User Control and GLG Library for C#/.NET

This is a native GLG C# Class Library for developing GLG applications using C# and .NET.

GLG ActiveX Control for C/C++ and C#/.NET on Windows

This is an alternative option for developing GLG applications in C/C++ and C#/.NET on Windows. It may be used to provide an OpenGL renderer option for C#/.NET applications.

Animating a Drawing

Once a drawing is displayed, it may be animated by setting parameters of the drawing using either resources or tags.

- **GLG Standard API** includes methods to set or query resources and tags defined in the drawing. These functions, along with a small assortment of convenience functions, are platform-independent, and can be used in both the Microsoft Windows and X Windows environments, C/C++/C#, .NET and Java.

Manipulating Objects in the Drawing at Run Time

The following GLG APIs extend the Standard API with additional functionality:

- **Intermediate API** extends the Standard API with methods for drawing introspection, such as traversing objects in the drawing, accessing objects' internals and custom properties, querying a list of resources defined in the drawing or in an individual object. It also includes methods for handling mouse interaction, object layout and advanced geometry manipulation, coordinate conversion, editing dynamics and constraints, and other methods that provide complete control over the objects in a GLG drawing.
- **Extended API** extends the Intermediate API with methods for programmatic object creation at run time. It is used to create or copy objects on the fly when the number of objects varies and is determined dynamically at run time. Other examples include dynamically configurable applications that create drawings based on a configuration file, or custom editor applications that need to create objects with the mouse.

GLG Widgets

The Toolkit provides a variety of prebuilt widgets, such as dials, meters, buttons and toggles, charts, avionics, process control and many other widgets. Each widget is a GLG drawing that can be loaded and customized in the GLG Builder. Custom widgets can be created by either modifying an existing widget, or created from scratch using the Enterprise Edition of the Builder.

In the Builder, widgets are available for drag and drop via the widget palettes. Custom palettes can also be integrated into the GLG Builder and HMI Editor.

Programming Tools

The GLG Toolkit includes several programming tools useful to application programmers. These include a test data generator, a file format converter, and a code generator for including a GLG drawing directly within an application’s code. The GLG Graphics Builder and file converter may also be used as scripting tools, for editing drawings using a script in batch mode, or to create new drawings using a script. Refer to the the *GLG Programming Tools and Utilities* chapter of the *GLG Programming Manual* for more information.

Chapter 2

Structure of a GLG Drawing

2

When you look at a GLG drawing, you see a collection of colored shapes that may be moving and changing according to a set of predefined instructions. This picture, however, is only the surface representation of the internal structure of a drawing. Internally, a GLG drawing actually consists of an arrangement of abstract data objects and the links between them. When you edit a drawing with the GLG Graphics Builder, or animate it with a program, you are accessing and modifying data within this data structure. The rendering of this elaborate internal structure into a simple two-dimensional image is the last and most superficial of the many steps that go into animating a drawing.

Of course, the two-dimensional image is not only what you see when the drawing is displayed, but also what you see when you create and edit a drawing. To understand what is really going on when you edit or operate a GLG drawing, however, you must understand the underlying structure of that drawing. This chapter introduces the basic elements of that structure.

Objects

The concept of the **object** is fundamental to GLG. Every component of a GLG drawing may be considered an object, and a drawing is nothing more than an arrangement of these objects and the links between them. The definition of a GLG object is similar to that of an object in C++ or some other object-oriented programming system. That is, an object is simply a thing that may be represented as a collection of data and the methods used to access that data. Some objects have a graphical appearance. For example, a polygon appears on the screen as a flat shape bounded by a line. Within the context of the GLG Toolkit, however, it is more useful to think of that polygon as a simple collection of points, colors, and line types (the data) and interfaces to access these data (the methods). Since the graphical appearance of the object is completely defined by its data, the program can control the way the object is rendered on the screen by changing these data using just a handful of methods common for all GLG objects.

An object's data are referred to as its **attributes**. Using the same example, a polygon's color, the width of its border lines, and the position of its vertex points, are each attributes of the object. Most of these attributes are stored as data objects, which keep the value, name and some other properties of each attribute. The notion of an attribute in GLG is recursive, as the attribute's value and name are attributes of an attribute object itself.

Some properties of an object are kept as plain data instead of using data objects. For example, the name attribute is kept as a character string to avoid infinite recursion of a name object having its own name attribute, and so on.

An object may contain other objects. An object that contains other objects is said to be their **parent object**. An object contained in some parent object is said to be its **child object**. We have already seen one example of this, since attributes represented by data objects are the children of the parent objects they describe. Other objects, such as the group, are designed to be containers of other objects. This arrangement of parent objects and their children defines the **object hierarchy**, about which more is said below.

In addition to a polygon, there are several other GLG **graphical objects**, such as circles, markers, viewports, and text objects. These are the objects that make up the visible aspect of a GLG drawing.

GLG differs from other drawing systems, however, in its assortment of **non-graphical objects**. You cannot see these objects, but they are a crucial part of the organization of the drawing. For example, a group object acts as a "container" for other objects, a series object defines a group of identical objects, a frame defines a set of anchor points to which other objects may be attached, and a transformation object contains directions for modifying the object to which it is attached.

Resources and Objects

To access an object at run-time, it has to be named. A named object becomes a **resource** which can be accessed by its name using the programming API. Resource mechanism provides a convenient way to manipulate objects in the drawing at run-time. In the Graphics Builder, the Resource Browser may be used to browse resources of the drawing.

The Attribute Object

An object is completely described by its **attributes**. A unique set of attributes defines a unique object. The particular set of attributes that make up an object is dependent on the type of the object. For example, a text object has a font, a color, a position, an angle, and so on, while a polygon has a fill color, a line width, and vertex positions. The term **property** is often used as a synonym for attribute.

Most of the attributes of a GLG object are themselves objects. In the same way that a polygon is a collection of positions, colors, and line widths, an attribute is also a collection of information. The information contained in an attribute object includes the data value of the attribute object, its data type and name. The type indicates whether the data value is a double-precision scalar number (signified throughout the GLG documentation by a **D**), a position in a three-dimensional coordinate system (**G** for "geometrical"), or a character string (**S**). The color values use **G** data type as well, since they represent coordinates in RGB color space.

The attribute's name is used to access the attribute as a **resource**. Named attributes become resources which can be accessed at run-time to supply dynamic data. In the Graphics Builder, the Resource Browser may be used to browse resources of individual objects or of the whole drawing. Each attribute also has a **default attribute name** which depends on the attribute type and may be used to access unnamed attributes. For example, the default attribute name of the polygon's line width is *LineWidth*. While attributes can always be accessed using their default attribute names, attribute names may be used to assign custom, application specific names to some attributes. For example, the *LineWidth* attribute may be given a custom name *box_width*. Attribute names take precedence over default attribute names, so it is a good idea to use names different from the default attribute names.

Note that there are several multi-valued data values used in a GLG drawing. Data flags, line types, font selections and so forth are all displayed as selections from a list of options. Internally, these are all **D** values, stored as the simple index to the table of choices. For binary attributes, where the value is TRUE or FALSE, or YES or NO, a zero value indicates false (NO) and any other value—usually

one—indicates true (YES). One exception to this rule is the *Visibility* attribute of an object. This attribute is a floating point value that can range from 0 to 1 and defines not only the object visibility (1-visible, 0-invisible), but also its transparency.

To avoid an infinite regression, the type, name and data values and some other attributes of the attribute object are not objects themselves. Among other such attributes are the *HasResources* flag, the *Global* flag, and the flag.

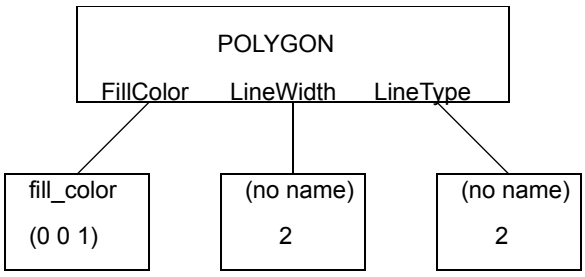
Resources and Attributes

GLG uses the terms **attribute** and **resource** to refer to the same object. There is, however, a distinction between the two that involves the method of access. An object such as a polygon has several attributes, and these are accessible by their default names. You can think of these as places within its data structure for data to modify an aspect of the polygon. For example, a blue polygon has a *FillColor* attribute that specifies its color is blue. This object consists of a data type (G), and a data value (0 0 1), and some housekeeping data.

The *FillColor* default attribute name is not an independent name. This name is not part of the attribute object's data. It only means something relative to the polygon to which it refers. For example, if a second polygon has its edge color attribute constrained to the fill color of the first polygon, the data object may be accessed either as *FillColor* of the first polygon or as *EdgeColor* of the second polygon. However, the attribute object can have its own name. If you name that object, it becomes a resource, and is accessible by its own name rather than only by its connection to the parent polygon.

Consider the diagram below. A polygon's data is depicted with three attributes: the *FillColor*, *LineWidth*, and *LineType*. (Of course a polygon has more attributes than this, but we have omitted most of them for clarity.) The *LineWidth* and *LineType* attributes are represented by identical objects; they are each given their meaning—and their default names—by where they are attached to the parent polygon object. The important point is that the default attribute names are recorded in the parent polygon, not in the attribute object data.

The *FillColor* attribute object, on the other hand, has both a default attribute name, given it by its position relative to the parent polygon, and its own name, *fill_color*, recorded in its own data. This attribute is a named resource, accessible independently from the parent polygon.



A Polygon and its Attributes

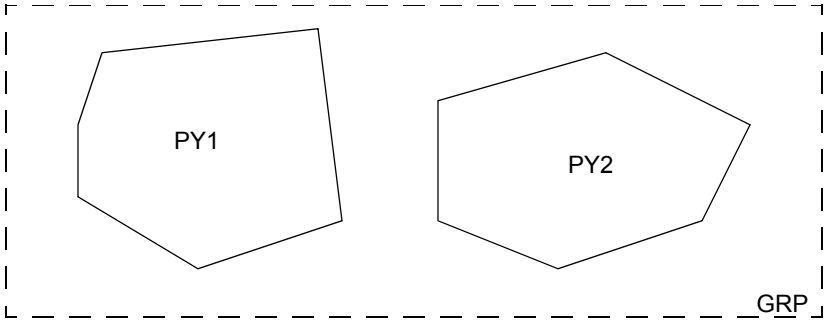
Supplying Data for Animation

Since everything in a GLG drawing is an object, and any attribute of an object can be accessed as a named resource, it becomes quite simple to change any aspect of a drawing by modifying its resources. For example, you can change the color of a polygon, rotate a group, move a viewport, even change the number of data samples in a graph by changing drawing resources. This is, in fact, the mechanism by which GLG drawings are animated. A program animates a GLG drawing by changing named resources with data from an external source.

Hierarchy of Objects

The arrangement of GLG objects in a drawing can be visualized as a hierarchy. Specialized container objects like groups and viewports are collections of disparate objects, while any GLG object can be said to contain its attribute objects.

Because a GLG object may contain other objects, the attributes of those secondary objects may be considered subsidiary attributes of the first object. This arrangement defines a drawing's **object hierarchy**. For example, consider a group object that contains two polygons.



A sample drawing of two simple polygons

Call the parent object *GRP*, and call the polygons *PY1* and *PY2*. Each polygon has an attribute called *FillColor* that indicates the color of the interior of that polygon. (This is part of the definition of the polygon object.)

You might also think of the *GRP* group as containing two "attributes," namely *PY1* and *PY2*. These objects in turn contain other attributes, like *FillColor*. In this case, the default name of the *FillColor* attribute of the *PY1* polygon would be *PY1/FillColor*. From a point outside the *GRP* group itself, the fill color of the *PY1* polygon would be called *GRP/PY1/FillColor*. In other words, there is a hierarchy of objects in a GLG drawing precisely comparable to the hierarchy of directory and file names used to denote the location of a file on a disk.

Hierarchy of Resources

The object hierarchy is a fixed representation of the parent-child relationships between objects in a GLG drawing. The *FillColor* attribute attached to *PY2* is always attached to *PY2*. In addition to the object hierarchy, however, a GLG drawing contains a **resource hierarchy** which represents relationships between the *names* of objects in a drawing. This hierarchy may or may not reflect the

underlying reality of the object hierarchy. This can be quite useful when you wish to shield a user from a very complex object hierarchy, or when it is conceptually simpler to address a drawing in terms defined by the resources. You typically use resources to animate a drawing.

An example will illustrate the use of having two separate hierarchies to represent the same objects. Using the objects in the drawing shown in the previous section, there is another way to consider the relationship between the two *FillColor* attributes and the *GRP* object. Suppose that the two polygons in the group are used to render two faces of a three-dimensional object. A cube, created by joining two-dimensional objects together at their edges, is an example of this technique. The object hierarchy would reflect the fact that this is a composite object, but we can construct the resource hierarchy so that it will appear to be a single object. In this case, one would want to specify one color for the entire object, rather than colors for each face, and the names *PY1* and *PY2* would likely not be useful for such an object.

In this case, to make the drawing act as you might expect, the color should be an attribute of the group. This way, the cube's color may be changed by accessing a resource called *GRP/fill_color*. Of course this would not reflect the reality of the object hierarchy, but you can arrange a GLG drawing this way by modifying the resource hierarchy. (We have here renamed the attribute for reasons explained below.)

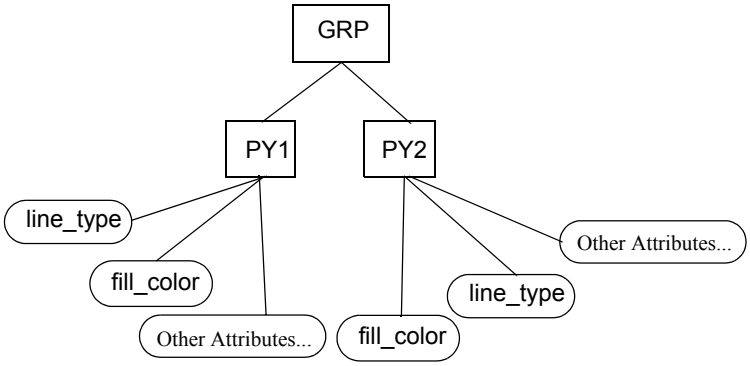
To make the resources of the child polygons appear to belong to the parent group, you can make the polygons **resource-transparent**. The resources of a resource-transparent object appear to belong to its parent. This way, the color attribute of the polygons will appear to be an attribute of the group itself.

Whether or not an object is resource-transparent is controlled by an attribute of that object called the *HasResources* flag. This flag indicates whether an object has resources of its own (*HasResources*=YES) or if its resources appear to belong to its parent (*HasResources*=NO). More specifically, a resource-transparent object's resources appear to belong to the first object above it in the resource hierarchy that is not itself resource-transparent.

Note that an object's default attribute names appear in the resource hierarchy as its children no matter if the object is resource-transparent or not. That is, even if *PY1* is resource-transparent, its *FillColor* attribute will appear under it. If we name the attribute *fill_color*, we can make it appear at different places in the resource hierarchy by modifying the resource-transparency of the parent object.

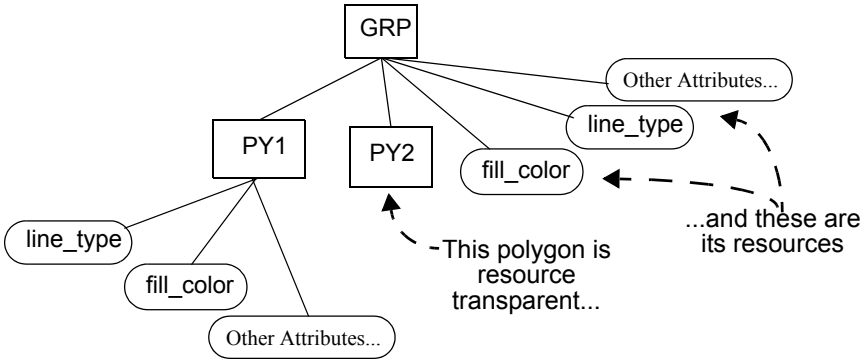
Since resource names are used to access objects, all resource names on the same level of hierarchy must be unique to avoid resource name conflicts. If two or more resources on the same hierarchy level have the same name, the first found resource object will be returned when the resource is accessed. Resources on different hierarchy levels may have the same names. For example, if both PY1 and PY2 polygons have HasResources=YES, each of the polygons may have its own *fill_color* resource accessible as *PY1/fill_color* for the first polygon and *PY2/fill_color* for the second one. Using such identical resource hierarchies is a powerful technique for drawings with a large number of similar objects.

The following diagram represents the hierarchy of objects in the drawing on page 28. If each object has resources (the *HasResources* flag is set to *YES*), and all objects are named, then the resource hierarchy is identical to this object hierarchy. Again, note that to avoid confusion about default attribute names (which always appear at the object level regardless of the *HasResources* flag), we have renamed the attributes shown in the diagram.



An Object Hierarchy

The following diagram shows the case where polygon *PY2* is resource transparent. Here, all the resources of the transparent polygon appear to be resources of the parent.



A Possible Naming Hierarchy

Note that the name of a resource transparent polygon is not used to access any of its named resources, but may still be used to access its named or unnamed attributes using their default names. Also, the default names of the *PY2* polygon attributes, such as *FillColor*, will still appear as resources of that polygon. This can be experimented with using the Builder's Resource Browser.

The *Alias* object may also be used to map a short logical name to an arbitrary resource hierarchy, defining custom shortcuts. Refer to the *Alias* section on page 140 for details. The Builder's Resource Browser provides controls to browse named resources, default resource names and aliases in any combination.

Organizing resources in hierarchies is a great way to simplify the complexity of a large drawing. When such drawing is browsed in the Builder using the Resource Browser, only the top-level resources will be visible on the drawing's level, and their sub-resources will be displayed only when a particular resource is selected. GLG does not have any limit for the depth of the resource

hierarchies, and each of the sub-objects in the drawing may have its own hierarchy of resources inside it, and so on. While the resource mechanism is very flexible and powerful, the application has to know the exact path of all resources it needs to access. Tags, described in the next chapter, provides an alternative data access mechanism with a flat hierarchy.

Tags for Database Connectivity

A data tag may be attached to any attribute or resource object for an alternative tag-based access to the resource value. In addition to accessing an attribute object by a hierarchical resource name, it may be accessed by global tags, which is convenient for applications that obtain the data from a database.

Tags are similar to resources and, same as resources, are used to update attributes with new data at run time. There is one important difference, though: while resources are hierarchical, the tags are global and their hierarchy is flat. All tags defined in the drawing are visible at the top level, which is very convenient for data connectivity with process databases. Tags provide a way to associate a name of the database field (the data source) with the attribute that needs to be updated with this data. In the Builder, the Tag Browser may be used to view tags of individual objects, as well as all tags defined in the drawing.

A tag object has *TagName* and *TagSource* attributes which facilitate database connectivity. The *TagSource* attribute contains the name of the database field that will supply dynamic data for updating the attribute. The *TagName* attribute defines a tag name which makes it easy for the user to persistently identify the tag regardless of the changes to its *TagSource*. To change the database mapping, the user can browse tags, select the tag whose database connection needs to be changed and edit its *TagSource*. The tag object also contains the *TagComment* attribute that may hold any user-defined information associated with the tag. If the attribute the tag is attached to has a range transformation, the tag editing dialog will also allow entering values for the *InLow* and *InHigh* parameters of the range transformation. The *TagAccessType* and *TagEnabled* attributes provide additional control over the use of a tag in an application. Refer to the *Tag* section on page 137 for more information.

At run time, the application will receive data change events from the database and will set the new value for tags defined in the drawing every time the corresponding database fields change. Since tags are global, the tag value can be set via its *TagSource* without the need to know any path names. If several attribute objects have tags with the same *TagSource*, all of them will be updated when a new tag value is supplied at run-time.

GLG provides API to supply data using both tags and resources. Refer to the *Tag-Based Data Access and Database Connectivity* chapter on page 60 for details of different ways of using tags for accessing data. Tags should be used for updating the drawing with data from a process database or other similar source. For other tasks, such as accessing objects in the drawing programmatically, resources provide a more flexible and powerful alternative.

In the Builder, the user can browse resources of a drawing and add tags to resources that need to be updated from a process database. For example, consider a drawing that has an object named *Valve1* which has resource named *Open* controlling the valve's open state, which needs to be updated from the process database field named *valve1_open*. The user can select the *Open* resource in the

Resource Browser, add a tag to it and set the *valve1_open* string as the value of the tag's *TagSource*. At run time, the application will receive data change events from the database and will set the new value for the *valve1_open* tag every time the *valve1_open* database field changes.

The Builder's Tag Browser may be used to examine tags attached to individual objects in the drawing, as well as display a list of all tags defined in the drawing. The Tag Browser has controls for sorting and filtering the tag list by either Tag Names or Tag Sources. The Tag Browser's *Unique Tag Sources/Names* toggle controls how tags with the same name are displayed. If it is turned on, only one tag will be displayed for each set of tags with the same *TagSource* or *TagName*; otherwise, all tag objects will be displayed, including multiple tags with the same *TagSource* or *TagName*.

The Tag Export and Tag Import features described in the *Tag Export and Import Features for Run-Time Tag Mapping* chapter on page 61 may be used for modifying all tags defined in the drawing. The tag import file may be used for run-time mapping of all tags defined in the drawing to a specific database by using the *GlgImportTags* API method. The *GlgCreateTagList* API method may be used in an application to query all tags defined in the drawing, as shown in the Tags Example source code.

Constraints

After you have made both the polygons in the diagram on page 28 resource-transparent, the colors of the faces are still accessed individually. True, we can now refer to the colors as attributes of the *GRP* object, but there are still two different polygons with two different colors. Worse, if the names have not been chosen well, there may exist name conflicts, where two different attributes have the same name.

Our goal is to create a complex object with a single color attribute (which may be accessed as a single resource), so the natural next step is to **constrain** the fill colors of the two polygons to be the same. This way, whenever the color of one is changed, the other one automatically changes as well. A constraint on some object's attribute is simply a requirement that the value of that attribute is always the same as the attribute of the other object.

We now have a group object called *GRP* that contains a single resource controlling the color of the entire complex object. You can now rename the group to have some more evocative name (for example, "cube"), and forget that it is, in fact, nothing more than a collection of simple graphical objects. Setting the "fill_color" attribute of this "cube" will change the color of all its faces.

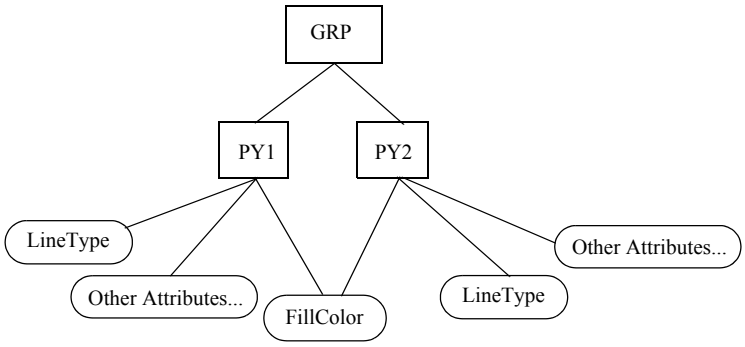
The Builder's *Edit All* group option provides an easy way to attach constraints to attributes of all objects in the group without repeating the constrain operation for each individual object. This is a convenient way to constrain the fill color attributes of all faces of the three-dimensional cube described above.

Constraints may be established between any two attribute objects of the same data type. These constraints connect one branch to another within the object hierarchy. Note that since GLG uses only three data types, a number of surprising constraints are possible. For example, you can constrain a polygon fill color to the position of a polygon vertex in 3-D space, since each is represented as a triple (geometrical) value.

Since point positions are simply object attributes, constraints may also tie one point of an object to another. Each vertex of a polygon is an object, whose value can be constrained to be the same as the value of a vertex of another polygon. You can use this method to construct arbitrarily complex three-dimensional shapes from the simplest components. The **constraints tracing** option described in the *Constraints Tracing* section on page 238 may be useful for debugging constraints defined in the drawing.

Note that there is no precedence or hierarchy associated with a constraint. When two attributes are constrained to each other, they behave as the same attribute object with the same name, value and flag settings. This means that one of the original objects, the one being constrained, and all its attributes (name, value, etc.) are thrown away. There is no concept of "master" or "slave" attributes; the resulting attribute may be accessed using any path in the resource hierarchy. For example, both *cube/PY1/FillColor*, *cube/PY2/FillColor* and *cube/fill_color* names may be used to access the constrained *fill_color* attribute of the above cube example.

In the following figure, the *FillColor* attribute of one of the polygons is constrained to the *FillColor* of the other. Now that the constraint has been established, there is no way to tell which object was thrown away and which kept. (Unless one of the original *FillColor* attribute objects was named, in which case, you can tell from the name of the remaining object. In general, the object you constrain is thrown away, and the one you constrain it to is kept.)



An Object Hierarchy Modified by a Constraint

Graphical Objects

In explaining the structure of a GLG drawing, we have made several references to **graphical objects**. The following sections describe the components of such an object. For a complete list and description of all the available graphical objects, see the *GLG Objects* chapter.

Note that while the set of atomic GLG objects consists of one- and two-dimensional objects such as lines and polygons, they are placed in a three-dimensional space within a GLG drawing. This can be the source of what initially seems odd behavior. For example, when you move one of the control points of a circle object, it may appear that the circle is being squashed into an ellipse. This is not the case; you are rotating it in space, and an ellipse is what you see when you look at a circle from off-center.

Attributes

Any graphical object has a set of default attributes that let you see the object rendered on a computer screen. A polygon's color, the width and hue of its border, its position in the drawing (whether it appears to be in front of or behind other objects), and whether it is open or closed, are among the attributes necessary to render that polygon. Most of the attributes in GLG are themselves objects and their editing dialogs in the Graphics Builder may be accessed by pressing the ellipsis button  positioned next to an attribute name in the Object Properties dialog.

A special attribute of any graphical object is its *Visibility*. This controls whether the object is rendered at all in the drawing. The ability to create invisible objects is useful for a variety of tasks. For example, it is usually desirable for a path polygon, which defines a track for another object to move along, to be invisible to the user. You can also use the visibility attribute to implement blinking objects or layers of objects that can be toggled on and off. The visibility attribute may be given values of 1 or 0 to make the object visible or invisible. It may also be set to a fractional value in the range of 0 to 1 to define transparency. For example, setting the visibility to 0.5 will result in an object which is half-transparent. On Windows, transparency is supported only with the OpenGL renderer.

Control points

A graphical object contains one or more **control points** that specify its shape and position. For example, a line is a polygon with two control points that specify the position of the line's ends. A point marker has only one control point, controlling its position. More elaborate objects have more control points. You can use these control points to vary the size and shape of the object. By constraining the control points of one object to the control points of another, you can construct complex drawings made up of many component objects, but controlled by a small number of points. The control points are themselves objects and their editing dialogs in the Graphics Builder may be accessed by *Shift*-clicking on them with the left mouse button.

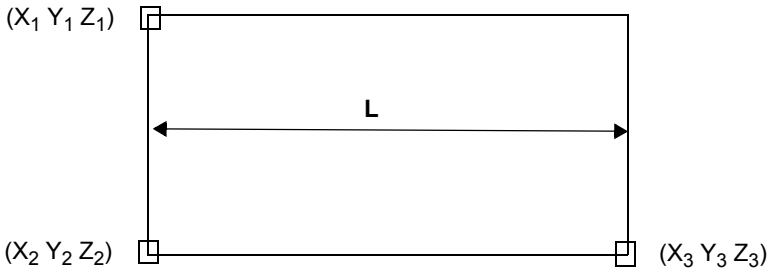
Note that an object's control points are sometimes distinct from the set of points that describe its boundary. For example, a parallelogram consists of three control points, defining the positions of three of its corners. The fourth point is generated dynamically from the others. Therefore, a parallelogram cannot be constrained by the fourth corner. (Although you can constrain other objects to that fourth point.)

Similarly, a circle object has two control points, which represent opposite ends of a vector perpendicular to the middle of the circle. You can use these control points to rotate the circle into a different plane from the one in which it was created. But what about the points that describe the circle's perimeter? These points are created dynamically by GLG, based on attributes of the circle object, such as its radius and resolution, as well as the positions of the control points. That is, they do not have a persistent existence as objects in the hierarchy. Therefore, it is impossible to constrain another object to them.

Transformations

By now, you understand how objects in a GLG drawing are built and constrained to each other, and how objects can be grouped together to make other objects. We have also introduced the concept of resources, by which object attributes are named—and can be changed—through the GLG editor or API. Using just these concepts, you can create and use sophisticated animated drawings.

GLG lets you control the shape of a graphical object not only by controlling its attributes, but also by attaching a **transformation** to it. This can provide a considerable savings in computation time and drawing complexity. For example, consider a rectangle defined by three control points. To control the length of the rectangle using the rectangle resources, you would have to directly control the position of at least one of the control points. This involves editing and setting at least one geometrical value, which consists of three spatial coordinates, as illustrated in the following figure:



Controlling the Length of a Rectangle

A simpler way to control the length of a rectangle is to create a resource that directly controls the length of the rectangle, and control that resource. This can be done by defining a *ScaleX* transformation and attaching it to the rectangle. The computational savings in this example may not seem great, but the advantages become more apparent when you consider an operation like rotating a ten point polygon around its center.

Several simple **geometric transformations** that may be applied to graphical objects, such as rotation, move, scale, and shear, can be defined with a matrix. The points of a graphical object, transformed with this matrix, produce the points of a new, transformed, graphical object. The details of the matrix construction are not important here, though they can be found in most introductory computer graphics texts.

There are two types of geometric transformations possible in a GLG drawing: static and dynamic transformations. They are sometimes called matrix transformations and parametric transformations, respectively.

GLG also provides transformations that can be applied to scalar and string data objects. The scalar transformations include various options for modifying a scalar (double) value, such as multiplication, division, range conversion, selecting a value from a list of values, etc. The string transformations are used for formatting the string text, displaying a scalar value as a text string and selecting a string from a list of strings.

Transformations as Objects

Like other GLG data, transformations are stored as objects. They are attached as child objects to the object they transform. There are geometrical transformations used to transform data objects of *G*-type, such as control points containing XYZ coordinates or color values containing RGB triplets, and non-geometrical attribute transformations used to transform double (*D*) and string (*S*) values.

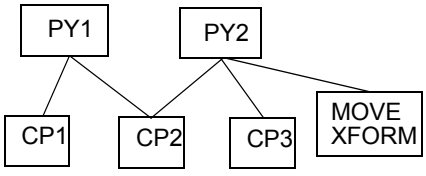
Several geometrical transformations can be attached to an object as a list of transformations. Before rendering an object in a drawing, GLG updates the coordinates of that object with whatever transformations are attached to it, and this produces rotations, shears, translations, and scale changes as specified in the list of transformations.

Note that a transformation may be attached to a control point *or* directly to a polygon or other drawable object. The GLG rendering engine processes the transformations attached to the control points first, using the transformations to convert the world coordinates of the control points into transformed values in the world coordinate system. The next step of the rendering process calculates the effect of all transformations attached to the drawable object, including any zooming transformations of the viewport and the world-to-screen transformation of the screen object. The resulting combined transformation is then applied to the transformed world coordinates of each object's control point to calculate the screen coordinates used for rendering.

The transformation attached to an object does not change the coordinates of its control points, but rather "projects" the object to appear in a place different from the position defined by the coordinates. If the transformation is attached to the control point, it also affects any other objects constrained to that point. However, if the transformation is attached to the object (such as a polygon, for example), then the transformation is applied to the polygon *after* the point position has been read by the rendering routines. While the transformation affects the polygon, it does not affect anything constrained to its control points.

This means that if two polygons have a constrained point and a move transformation is attached to one of polygons, only the transformed polygon moves, as the transformation "projects" it into a different position. The transformation doesn't change the coordinates of the constrained control point that is the child of both polygons, and the other polygon remains in place. Any changes to the constrained point will still affect both polygons: if the constrained point is moved (by the mouse, for example), both polygons will move but each in its own "projected" space.

The following diagrams illustrate the situation. In the figure below, two polygons are constrained at a control point CP2. A move transformation is applied to *PY2*, which moves it somewhere.



When the move transformation is applied, Polygon *PY2* moves, but Polygon *PY1* does not, even though one of its control points is constrained to a point of *PY2*, since the coordinates of the control point are unaffected by the transformation (see *Fig.1* and *Fig.2* below). The transformation affects only *PY2* by projecting it into a different position without changing the control point. If the control point CP2 moves (*Fig.3*), both polygons move, each in its own space.

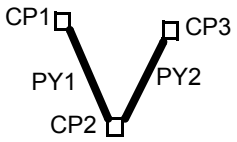


Fig.1: Original position

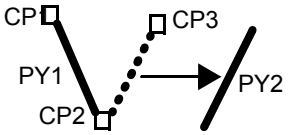


Fig.2: Attaching transformation

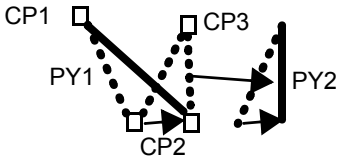
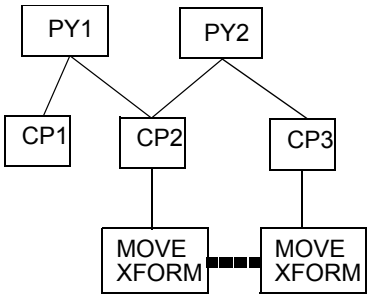
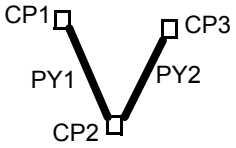


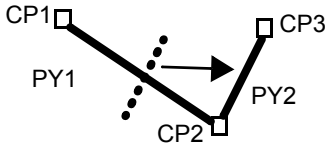
Fig.3: Moving constrained point

To apply a move transformation to a polygon in a way that will make all the constrained points to respond to the move transformation, apply constrained copies of the transformation to the control points themselves, as in the diagram below.



Now, when *PY2* is moved by the mean of changing move transformation's parameters, as shown below, it will drag along part of *PY1*. To move all of *PY1* with *PY2*, the same transformation could be attached to the *CP1* control point.





An alternative way to move both objects together would be to create a group containing the two polygons and apply the transformation to that group. You could also attach a constrained copy of the *PY1* transformation to the *PY2* object.

A constrained transformation is a transformation of the same type as the original transformation and with all parameters constrained to the corresponding parameters of the original transformation object. It may be created in the Builder by marking the original transformation with the *Mark Object* button of the transformation properties dialog and then using the *Use Marked* transformation option when attaching a new transformation. The Builder's *Attribute Clone Type* option must be set to *Constrained Clone*, which is the default value, to create constrained transformations.

Static Transformations

The simplest kind of geometric transformation is the **static transformation**. This transformation type contains the actual matrix used to transform the parent object, and is therefore sometimes called a **matrix transformation**. Before the parent object is rendered, its defining points are transformed using this matrix.

The disadvantage to the simplicity of this transformation object is its lack of flexibility. Once you create a static transformation object, editing it is cumbersome, although it may be easily combined with other static transformations or deleted. If a static transformation attached to some object defines a move of 50 units in the X direction, changing the transformation to move the parent object 75 units involves either combining it with a new static move transformation of 25 units, or discarding the 50-unit move object and creating an entirely new 75-unit move.

In the Builder, when a static transformation is attached to an object which already has a static transformation, both transformations are merged, and the object's single static transformation is modified to contain the resulting combined matrix.

Naturally, while useful for creating and editing graphical objects in drawings, this sort of transformation is not very useful for animation.

Transforming Object Points

Instead of attaching a transformation to an object as a child, the transformation may be used to physically change the coordinates of object's control points. In this case, no transformation is attached to the object, and instead the transformation is used to recalculate coordinates of the objects' control points.

For example, when an object is moved, stretched or rotated with the mouse in the Builder, the corresponding transformation is used to change the object's points. The Builder also provides the *Transform Object Points* option which may be used to change object's points by applying user-defined transformations. The GLG API also provides methods for transforming both an individual control point and all points of an object with a user-supplied transformation.

The object's flag controls the way the transformation is applied using *Transform Object Points*. If it is not set, the coordinates of the object's control points are changed. If the flag is set, the transformation will be attached to the object instead of transforming it's point coordinates when the object is moved, stretched, or otherwise transformed, both in the Builder and programmatically.

Dynamic Transformations

In order to animate a transformation, you must use a **dynamic transformation**. This type of transformation stores various parameters used to generate the transformation matrix on the fly, which are then used to transform the parent object. For example, the object that defines a rotation transformation contains the center of the rotation (a geometrical value), and an angle. Before rendering the parent object, the center and the angle are used to generate a rotation matrix, which is then applied to the parent. If the center or angle parameters are changed, the rotation matrix will be recalculated, changing the position of the object the transformation is attached to. If the angle

parameter is named, an application program can change its value at run time by accessing the angle parameter as a named resource. This will result in animating the object the rotate transformation is attached to. It will rotate according to the changing angle value.

Since this style of transformation uses certain parameters to generate the transformation matrix, it is sometimes called a **parametric transformation**.

Geometrical dynamic transformations typically use two numbers multiplied together to specify the effective value to use for its main parameter. For example, a rotation transformation uses an "angle" and a "factor" to define the rotation angle (the effective angle value used for rendering will be the product of these two parameters). This lets you control the input range of the controlling parameter.

For example, you could set up a rectangle that rotates through 90 degrees by setting the angle to 90, and letting the factor range from zero to one to accommodate an application which defines the maximum rotation angle of 90 degrees and then supplies a normalized value in the range of 0 to 1 to control the rectangle's rotation within this 90 degree span. The [0;1] range of the factor may be changed by attaching a range transformation to the factor attribute object itself.

If an application wants to control the rotation by supplying the rotation angle instead of a normalized value in some range, the factor of the rotate transformation may be set to 1: then the angle value will control the rotation directly. A common mistake is setting the factor to 0, in which case changing the angle value will not do anything, since the product of the two numbers will still be equal to 0.

For a complete list of the available dynamic transformations, see the *Transformation Object* section of the *GLG Objects* chapter.

Alarms

An alarm object can be attached to any data or attribute object to monitor its value. Internally, alarms are implemented as a special type of a transformation object whose only action is to produce an alarm message when the alarm conditions are met. Refer to the *Alarm Object* section of the *GLG Objects* chapter for a list of available alarm objects.

The View

A GLG graphical object exists in its own three-dimensional coordinate space. In order to draw a picture of that object on your two-dimensional screen, a mapping must be established between the object and the screen. Because an object can consist of several other objects, each with its own coordinate system, the mapping entails several steps. The steps involved in the mapping are also referred to as **modeling transformations**.

Coordinate Systems

The graphical objects in a GLG drawing are all defined by points in three-dimensional space. The coordinates of any point indicate a position for that point relative to some origin. It is possible, however, that two different points can be defined relative to different origins, and that their coordinates can have altogether different meanings.

The position of the origin, the units, and the directions of the unit vectors all define a **coordinate system**. Components of a single drawing can refer to several different coordinate systems, each related to another through a series of implicit and explicit transformations.

There are several layers of geometrical transformations that affect the way the object is rendered. The transformations may be attached to the object's control points, to the object itself, as well as to any of the object's parents. In addition, there may be user-defined viewing transformations attached to the viewport, as well as internal transformations that control the viewport's integrated zooming and panning. Finally, there is a world-to-screen transformation used by a screen object to map the world coordinate system of the drawing to the changing viewport size.

Each transformation changes the coordinate system by transforming all objects affected by it according to the transformation type, thus defining a new coordinate system. In GLG, the following types of coordinate systems are defined:

Parent Coordinates

The **parent coordinate** system defines the position of the origin relative to the object's parent, and the orientation of the parent in three-dimensional space. The parent coordinate system includes a combined effect of all transformations attached to the object's parent as well all its grand-parents.

In the editor, the parent coordinate system is used by default when the object is created. Consider an example when a circle centered about the origin *(0,0,0)* is created inside of a group. If the group is transformed by a move transformation with a move vector *(100,100,100)*, the origin of the circle will appear at *(100,100,100)* instead of *(0,0,0)*.

If none of the object's parents have transformations attached to them, the parent coordinate system coincides with the drawing coordinate system.

*Object Coordinate*s

Since any GLG object can have a transformation attached to it, its coordinate system may be different from its parent coordinate system. The **object coordinate** system includes a combined effect of all transformations attached to the object as well as all transformations attached to all of its parents. If the object does not have any transformations attached to it, its object coordinate system coincides with its parent coordinate system.

In the Builder, the coordinate values of the object's control points are entered with respect to the object coordinate system. The rendered position of each control point in the drawing is then defined by all transformations attached to the object. Alternatively, the point position can be entered in any of the defined coordinate systems and the point coordinates in the object coordinates system will be automatically calculated and stored in the control point object.

If an object does not have any transformations attached to it, the object coordinate system coincides with the parent coordinate system.

Drawing Coordinates

The **drawing coordinate** system is defined by the particular view you have of a drawing. The **main view** of a drawing is defined to be the one where the Z axis is pointing directly at the viewer, the X axis is level and pointing to the right, and the Y axis is pointing up. The view coordinates are defined by the matrix transformation that rotates, zooms, and moves this main view into the view of the drawing that is actually seen. The drawing coordinate system includes the combined effect of any viewing transformations attached to the viewport, the viewport's integrated zooming and panning transformations and the world-to-screen mapping transformations of the screen object.

If the viewport does not have any viewing or zooming and panning transformations, its drawing coordinate system coincides with the world coordinate system.

World Coordinates

The world coordinate system includes only the effect of the world-to-screen mapping transformations of the screen object.

The viewport's world coordinate system depends on the settings of the *Resizable* attribute of the viewport's screen. If the viewport's *Resizable* attribute is set to YES (WORLD), the world coordinate system's origin is always positioned at the center of the viewport regardless of any scrolling and panning, and the extent of the visible portion of the viewport in the world coordinates is [-1000;+1000] by default. The default extent may be changed by using the *SpanX* and *SpanY* attributes of the viewport's screen object. The exact coordinate mapping is also affected by the settings of the screen's *Stretch* and *PushIn* attributes. The X axis points to the right, the Y axis points up, and the Z axis points to the viewer. When the viewport is resized, objects in the drawing are resized with the drawing area.

If the *Resizable* attribute is set to NO (SCREEN), the world coordinate system is the same as the screen coordinate system, with the origin at the upper left corner of the viewport's window and the X and Y axes pointing to the right and down respectively. The Z axis points away from the viewer to maintain a right-hand coordinate system used for 3D rendering. The coordinate mapping does not change when the viewport's window is resized, keeping the size of the objects in the drawing constant.

If the *Resizable* attribute is set to NO (GLG SCREEN), the world coordinate system has the origin at the upper left corner of the viewport's window as well, but the X and Y axes are directed to the right and up to match the direction of the axes in the default WORLD coordinate system. The coordinate mapping does not change when the viewport's window is resized, keeping the size of the objects in the drawing constant.

If the *Resizable* attribute is set to NO (SCREEN CENTER), the world coordinate system's origin is always positioned in the center of the viewport's window, and the X and Y axes are directed to the right and up respectfully. When the viewport's window is resized, the coordinate mapping changes to keep the origin in the center of the window, but the size of the objects in the drawing is kept constant.

When a new widget is created, the *Resizable* attribute of the widget's viewport is set automatically depending on the selected *Stretch/Resize* option for creating a new widget. For example, the GLG SCREEN setting is used for a widget created using the *File, New, Widget (Fixed Scale)* menu option.

Refer to the the *Screen* section on page 93 for more information on the *Resizable* attribute.

GIS Coordinates

The **GIS coordinate system** is a special type of the world coordinate system used to render objects on top of the map in the GIS Rendering Mode. In this mode, X and Y coordinates are specified using the GIS longitude and latitude, respectively.

The GLG Toolkit provides an integrated GIS Object for rendering GIS maps in the GLG drawing. Any graphical object added to the GIS Object will be displayed on top of the map using the GIS Rendering Mode, providing a convenient way to position dynamic objects on the map using lat/lon coordinates.

Screen Coordinates

The **screen coordinate** system uses the screen coordinates of the native windowing system. Its origin is always located in the upper left corner of the viewport.

The final mapping transformation takes the drawing from an internal representation of three-dimensional space into a two-dimensional one. The origin of this coordinate system is defined to be the upper left corner of the window and the horizontal and vertical extent of the visible part of the viewport's window are equal to the window's width and height, respectively.

Lighting

In order to see an object in three-dimensional space, there must be a source of light to illuminate it. The GLG system provides two different sources of lighting: **illumination** and **ambience**.

Illumination is the simplest form of lighting to understand, because it is the most obvious analogy to what we observe in the real world. Illumination is the light cast by a point light source in a certain direction. For example, a light source on the right side of some object leaves the left side of that object in shadow. Each viewport in a GLG drawing can have a light source to shine in it.

The viewport uses a *Light* object to store all lighting attributes. This allows for space for viewports that do not use lighting and 3D shading to be conserved. The *Light* object has attributes that allow you to position and aim the light.

The other form of lighting used in a GLG drawing is ambient light. Unlike illumination, ambient light is cast evenly on all objects, no matter which direction they face. A *Light* object's attributes let you determine the proportion of light cast by illumination and ambience.

The lighting facility provided with GLG is a limited one; it is useful for dynamic visualization of three-dimensional objects, but is not really appropriate for photo-realistic rendering a scene. For example, a user may position the light, but the objects do not cast shadows on other objects.

A polygon's *Shading* attribute provides additional control over shading of individual polygons in the viewport.

Input Handlers

GLG provides a variety of **input handlers** used to build input objects, such as buttons, toggles, sliders, knobs, etc. An input handler is an object that processes input events and converts them into the corresponding resource changes, depending on the type of the input handler. For example, a slider input handler (***GlgSlider***) converts the mouse move and mouse click events into value changes of the resource that controls the slider knob's position.

An input handler is attached to a viewport by entering the handler's name into the viewport's *Handler* attribute (the viewport's *DisableInput* attribute must be set to NO to enable the handler). Each handler type implements predefined logic associated with a particular input object. For example, the *GlgButton* handler implements the input logic of the push button and toggle widget.

Each handler searches for some predefined set of resources to control in the viewport they are attached to. For example, the *GlgKnob* handler searches for the resource named *Value* and then changes its value to rotate the knob when it is moved with the mouse.

The knob widget is designed in such a way that the *Value* resource controls the angle of the knob's rotation, and changing this resource from 0 to 1 moves the knob from the start angle to the end angle. The handler always changes the resources it controls in the range of 0 to 1. A range transformation may be used to convert the default range of 0 to 1 to a different range.

The behavior of an input handler may be modified by defining certain resource names it recognizes. For example, the *GlgButton* handler searches for the resource named *OnState*. If this resource is found, the handler implements a toggle. Otherwise a push button is implemented.

An input handler searches for resources at the top level of the viewport hierarchy, so any controlling resources must be visible at the viewport's top level. **Alias** objects may be used to make resources defined inside the hierarchy be visible at the top level. It is also important to set the *HasResources* flag for the viewport when adding a handler to it. Without it, the viewport will be resource-transparent, and all its resources will "leak" through and will be visible at the level of its parent, instead of the viewport's level.

When an input handler processes input events and changes the values of resources it controls, it also generates messages which are sent to the application program via the **Input callback**. This lets the program react to the user input depending on the message's **Action** type. Refer to the *Callback Events* chapter of the GLG Programming Reference for details.

For input objects that have corresponding native widgets available (buttons, toggles, sliders and text inputs), GLG provides two types of handlers: one for the graphical version of the widget using GLG graphical objects to implement the widget's graphics, and another for use with the native widget to implement the input object. For example, GlgButton may be used with custom graphical buttons and toggles, and GlgNButton may be used with native Motif, Windows or Java buttons and toggles.

Integrated Features of the GLG Drawing

The GLG drawing provides integrated support for zoom and pan, object tooltips, mouse over highlight, integrated input, selection and custom selection events. These integrated features greatly simplify application development. Refer to the *Integrated Features of the GLG Drawing* chapter on page 45 for details.

Chapter 3

Integrated Features of the GLG Drawing

3

Integrated Zooming and Panning

All viewport objects support integrated zooming and panning. The panning (or scrolling) is controlled by the viewport's *Pan* attribute. If panning is enabled, the viewport displays integrated scrollbars that allow the drawing to be scrolled when it extends beyond the boundaries of the viewport's visible area. The *Pan* attribute allows the user to enable X and Y scrollbars independently, to scroll the drawing area in only one direction, or in both directions. The *Pan* attribute also provides settings for displaying the scrollbars only when needed, so that they appear only when the drawing extends outside of the visible area and may need to be scrolled. In addition to using the scrollbars, the drawing can also be **scrolled by dragging it with the mouse**.

Integrated zooming includes support for *Zoom In, Zoom Out, Zoom To* and other zoom actions. At run time, the integrated zooming and panning can be performed by a single call to the *GlgSetZoom* method described on page 89 of the *GLG Programming Reference Manual*.

If the viewport's *ZoomEnabled* attribute is set to *YES*, the viewport object also handles keyboard accelerators for zooming and panning. Refer to the *Viewport* section on page 84 for the complete list of keyboard accelerators. If *ZoomEnabled* is set to *NO*, the accelerators are disabled, but the integrated zooming and panning is still available at run time via the *GlgSetZoom* API method. Performing zoom and pan operations on a viewport generates *Zoom* and *Pan* events which can be handled in a program. Refer to the *Appendix B: Message Object Resources* section on page 348 of the *GLG Programming Reference Manual* for more details.

There are several zoom modes. In the default Drawing Zoom Mode, zooming and panning operations zoom and pan the objects displayed in the drawing. In addition to the default zoom mode, there are specialized zoom modes for zooming and scrolling charts and maps, as described below.

Chart Zooming and Scrolling

The Chart Zoom Mode is used to zoom and scroll data plotted in real-time charts. In the Chart Zoom Mode, the zoom and pan operations zoom and scroll data plotted in the chart, and integrated scrollbars scroll the chart in the horizontal or vertical direction. The chart can also be scrolled by dragging it with the mouse. Zooming to an area of the chart with the mouse is also supported.

At run time, the integrated zooming and panning can be performed by a single call to the *GlgSetZoom* method described on page 89 of the *GLG Programming Reference Manual*.

The Chart Zoom Mode of a viewport can be set in the Builder by using the *Arrange, Chart Zoom Mode, Set as Parent Viewport's Chart Zoom Object* menu option while the chart object is selected, or by using the *GlgSetZoomMode* API method at run time. The Chart Zoom Mode is preset for all charts in the *Real-Time Charts* palette.

Zooming and Panning GIS Maps

The GIS Zoom Mode is used for zooming and panning a map displayed in the GIS Object. In the GIS Zoom Mode, the zoom and pan operations zoom and pan the map displayed in the GIS object,

and integrated scrollbars scroll the map vertically or horizontally. The map can also be dragged with the mouse. For example, a globe displayed in the orthographic projection may be rotated by dragging it with the mouse. Dragging complex maps may require a fast CPU.

The GIS Zoom Mode of a viewport can be set in the Builder by using the *Arrange, GIS Zoom Mode, Set as Parent Viewport's Chart Zoom Object* menu option while the GIS object is selected, or by using the *GlgSetZoomMode* API method at run time. The GIS Zoom Mode is persistent and is saved with the drawing.

Accessing Resources of Integrated Scrollbars

When integrated scrollbars are activated, they can be accessed as viewport resources using resource names *GlgPanX* and *GlgPanY*. There is also an object named *GlgPanSpacer* used in the lower right corner of the viewport when both scrollbars are enabled. By default, the colors of the scrollbars are constrained to the color of the viewport. To change them to be different from the viewport's color, unconstrain the *FillColor* attribute of the viewport, then set the scrollbar's *FillColor* to a different color value.

The integrated scrollbars automatically adjust the size and position of their knobs to indicate the portion of the drawing visible in the viewport window. Since GLG has an infinite coordinate system, the drawing is not limited to any page size, and the scrollbars use the extent of all objects in the drawing as the drawing size. If some objects in the drawing are partially or completely clipped out by the viewport window, the thumbs will reflect the portion of the drawing visible in the window and will allow the user to scroll the drawing. If all objects in the drawing are completely visible in the window, the thumbs will indicate that there is no need to scroll.

If it is desired to define a fixed drawing size or a maximum scrolling extent that is bigger than the extent of objects in the drawing, it may be easily accomplished by adding a rectangle to the drawing to serve as a drawing boundary. The rectangle will provide a visual indication of the drawing extent, or can be made invisible if required. The scrollbars will use the extent of the rectangle as the drawing size, allowing the user to scroll up to its extent.

Using Custom Scrollbars

By default, the integrated scrollbars use native scrollbars that differ in appearance on different platforms. GLG scrollbars may be used to provide the same look and fill regardless of the platform by setting the *GlgNativeScrollbars* global configuration resource to 0. The *GlgNativeScrollbars* global configuration resource is automatically set to 0 for Qt and GTK integrations.

When *GlgNativeScrollbars* is set to 0, the application may also provide its own custom scrollbars to use as the integrated scrollbars. This is done by setting the *GlgVScrollbarRef* and *GlgHScrollbarRef* global configuration resources to the filenames (or URL when used on the web) of the drawings that contain the widgets to be used as the vertical and horizontal scrollbars.

Integrated GIS Object, GIS Rendering and GIS Editing Mode

The GIS maps can be embedded into the GLG drawing via an integrated GIS Object. The GIS Object displays a map image in a selected GIS projection and transparently handles all aspects of interaction with the GLG Map Server, automatically issuing map requests every time the map is resized, panned or zoomed.

The GIS Object can be used as a container that holds dynamic icons, polylines and other graphical objects that need to be drawn on top of the map. The objects are drawn on the map in the **GIS Rendering Mode**, in which the coordinates of the objects' control points are interpreted as lat/lon coordinates. This allows positioning of icons and lines on the map by defining their lat/lon coordinates directly, without any coordinate conversions. When the map is zoomed or panned, the objects on the map will be automatically adjusted to stay with the map, with no application support required in the previous releases of the Toolkit.

The Graphics Builder supports the **GIS Editing Mode** for creating and editing objects to be drawn on top of the map. In this mode, dynamic icons, polylines and other objects can be drawn and positioned on the map with the mouse in the lat/lon coordinates. The Builder automatically converts the mouse position to lat/lon coordinates, which are stored in the object's control points. The Builder transparently handles GIS projections, which allows the user to draw polylines on top of the globe displayed in the ORTHOGRAPHIC projection.

To start the GIS Editing Mode, select the GIS Object, then press the *Hierarchy Down* button to go down into it. In the GIS Editing Mode, you can draw and position objects on top of the map with the mouse, as well edit attributes of the previously created objects. All objects added to the GIS Object for the GIS rendering will be contained in its *GISArray* and will be saved with the GIS Object. Dynamic icons and other graphical objects may also be added to the GIS Object programmatically at run time using the *GlgAddObject* methods. The GLG GIS Demo and GLG AirTraffic Control Demo may be used as source code examples of adding dynamic icons at run-time.

The map displayed in the GIS Object may be zoomed and panned using the integrated zooming and panning features, including integrated scrollbars. The map may also be dragged with the mouse as described in the *Integrated Zooming and Panning* chapter above.

The GIS Object supports GIS queries which can be used to obtain information about the GIS feature under the current cursor position. The GIS query returns a complete list of attributes of the GIS feature, which may include city or road name, city population, road type, and other application-specific GIS attributes.

Refer to the the *GIS Object* chapter on page 80 for more information.

Integrated Tooltips

GLG drawings supports integrated object tooltips, as well chart and axis tooltips. Custom tooltip formatters can also be used for generating context-dependent tooltip strings.

Object Tooltips

A tooltip action may be added to any object via the *Object, Tooltip, Add Tooltip* menu option in the Enterprise edition of the GLG Builder. A tooltip will be displayed when the mouse hovers over the object.

Old-type (prior to v. 3.5) tooltips are enabled by adding the *TooltipString* resource to an object.

The first found tooltip action of the object or its nearest parent is used to display the tooltip. For example, if the object is a group, the tooltip will be displayed every time the mouse moves over any object in that group. If the object inside the group has its own tooltip action, that object's tooltip action will be used to display the tooltip instead of the group's tooltip action.

To activate processing of object tooltips in the viewport, the viewport's *ProcessMouse* attribute has to include the *Tooltip* mask. If it is set to *Tooltip (Named Objects)*, tooltips will be activated for all named objects, and the object's name will be used to display the tooltip instead of the object's tooltip action.

The *Run* mode may be used to prototype the tooltips in the Builder. At run time, tooltips are handled automatically, with no application code required.

The *Tooltip* event is generated every time an object tooltip is activated or erased. Refer to the *Tooltip Message Object* section on page 364 of the *GLG Programming Reference Manual* for more details.

In the C# environment, balloon tooltips can be enabled by setting the *GlgNativeTooltip* global configuration resource described on page 342 of the GLG Programming Reference Manual.

Chart and Axis Tooltips

Chart and axis objects support integrated tooltips displayed when the mouse hovers over a chart or an axis object. The tooltip displays information related to the mouse position over the chart or the axis.

For a chart object, the tooltip contains information about the data sample selected by the current mouse position. If the mouse hovers over the chart's X or Y axis, the tooltip displays the time or the Y value corresponding to the mouse position.

For an axis object, the tooltip converts the mouse position to a corresponding axis value and displays it in the tooltip.

The content of the tooltip is controlled by the chart's and axis's *TooltipFormat* attribute, see page 116.

The chart tooltip is defined by a tooltip action with the *Tooltip= $ChartTooltip* attached to the chart object, and *Tooltip=$AxisTooltip* is used to attach a tooltip action to an axis object.

Custom Tooltip Formatters

A custom tooltip formatter can be set using the *GlgSetTooltipFormatter* function. A custom tooltip formatter is invoked every time a tooltip is activated and provides a string that will be displayed in the tooltip. This enables an application to generate dynamic context-dependent tooltips at run time.

Tooltip Colors and Appearance

Tooltip colors are controlled by the *GlgTooltipLabelColor* and *GlgTooltipBGColor* global configuration resources described in the *Appendix A: Global Configuration Resources* chapter of the *GLG Programming Reference Manual* on page 341. If these resources are not set, the default colors inherited from the respective run-time environment are used to render a tooltip.

Multi-line tooltips are supported: if a tooltip string contains line breaks, it will be rendered as a multi-line tooltip. The *GlgTooltipTextAlignment* global configuration resource may be used to specify alignment of rows in a multi-line tooltip. By default, the rows are aligned to the left of the tooltip box.

Integrated MouseOver and MouseClick Actions

Various *MouseClick* and *MouseOver* actions can be added to an object in a drawing using an integrated action object. The following describes available action types and examples of their use. Refer to the the *Action Object* chapter on page 176 for a detailed description of the action object's attributes.

MouseOver Highlight

The GLG drawing supports object highlighting when the mouse is moved over the object. The object highlight on mouse over may be used to create a "hot spot". Moving the mouse over such a hot spot highlights the object and, using either custom events or commands described below, sends a message to the program to perform application-defined actions.

The mouse over highlight is implemented by attaching an action to the object using the *Object, Actions, Add Mouse Feedback* menu option of the Enterprise edition of the Builder, then setting the actions' *ActionType=TRACE_STATE* and *Trigger=MOUSE_OVER*. The action will change the value of its *State* attribute to 1 when the mouse moves over the object, and resets it to 0 when the mouse moves away.

The action's *State* attribute may be constrained to any attribute of the object or any of the transformations attached to it. For example, State is constrained to the *LineWidth* attribute or a polygon, the line width of the polygon will be set to 1 when the mouse moves over it, displaying the polygon's edge. When the mouse moves away from the polygon, the polygon's line width will be reset to 0 and the polygon's edge will disappear.

Consider another example with a list transformation attached to the polygon's *LineWidth* attribute. If the list contains two elements with values 1 and 3, and *State* is constrained to the *ValueIndex* parameter of the list transformation, then the polygon's line width changes from 1 to 3 when the mouse moves over the polygon and changes back to 1 when the mouse moves away. Visually, this will create a highlighting effect. Various other types of highlighting may be used. For example, the object's color may be changed by attaching a color list transformation to the object's color attribute. If more than two attributes need to be highlighted, they may be constrained to change together.

To enable processing of the *MouseOver* events, the viewport's *ProcessMouse* attribute has to include the *Move* mask. An action's *ProcessArmed* attribute may be used to activate the action only when the *Control* key is held down.

In the GLG Builder, the *Run* mode may be used to prototype the *MouseOver* highlight. At run time, it will be handled automatically, with no actions required in the program.

An old-style (prior to v. 3.5) mouse over highlight implemented by naming an object's resource *MouseOverState* is also supported for backward compatibility. The value of the object's *MouseOverState* resource is set to 1 when the mouse moves over the object and to 0 when it moves away.

The *MouseOverState* resource must be visible at the proper place in the object hierarchy. If a polygon in the previous example has *HasResources* set to *YES*, the *MouseOverState* resource will appear as a resource of the polygon, and the polygon will be highlighted when the mouse moves over it. If the polygon is part of a group and the group's *HasResources* is set to *YES*, but the polygon's *HasResources* is set to *NO*, the *MouseOverState* resource will appear as a resource of the group and the polygon will be highlighted every time the mouse moves over any object in the group.

MouseClick Feedback and Toggle

An object in the drawing may be set up to provide visual feedback when it is pressed with the mouse button, for example to "depress" an object on a mouse click. A mouse click toggle functionality may be used to alter the object's visual appearance every time it is pressed with the mouse button.

The mouse click feedback and toggle may be used to implement lightweight viewport-less buttons and toggles, which provide a visual feedback on a mouse click and, using custom events feature described in the next section, send a message to the program when the button is pressed. In a regular button input object, an input handler attached to the button's viewport handles the button's visual feedback.

The mouse click feedback is implemented by attaching an action with *ActionType=TRACE_STATE* and *Trigger=MOUSE_CLICK* to the object using the *Object, Actions, Add Mouse Feedback* menu option of Enterprise edition of the Builder. The action will change the value of its *State* attribute to 1 when the object is clicked on with the mouse, and resets it to 0 when the mouse button is released. The action's *MouseButton* attribute defines the mouse button that activates the feedback.

The action's *State* attribute may be constrained to any attribute of the object or any of the transformations attached to it. For example, *State* may be constrained to the *Factor* attribute of a move transformation attached to the object. When the object is clicked on with the mouse, the value of the transformation's *Factor* will be set to 1 and the object will move, showing visual feedback. When the mouse button is released, the *Factor* will be reset to 0 and the object will move back.

To toggle the object's state on a mouse click, an action with *ActionType=TOGGLE_STATE* and *Trigger=MOUSE_CLICK* may be used. The action will toggle the value of its *State* attribute every time the object is clicked with the mouse. If the *State* is constrained to the *Visibility* of an object in the drawing, the object's visibility will alternate every time it is pressed with the mouse. It is easy to imagine many various ways to implement a custom toggle object using this functionality.

The *SET_STATE* and *RESET_STATE* values of the action's *ActionType* attribute may be used to set the value of the action's *State* attribute to 1 or reset it to 0. For example, two objects in a drawing may be used to start or stop animation attaching an action with *ActionType=SET_STATE* to one object, and an action with *ActionType=SET_STATE* to another object. The *State* attribute of both actions may be constrained to the *Enabled* attribute of a timer that drives animation in the drawing.

To enable processing of the *MouseClick* events, the viewport's *ProcessMouse* attribute has to include the *Click* mask. An action's *ProcessArmed* attribute may be used to activate the action only when the *Control* key is held down.

In the Builder, the *Run* mode may be used to prototype the *MouseClick* feedback and toggle behavior. At run time, they will be handled automatically, with no actions required in the program.

An old-style (prior to v. 3.5) mouse click feedback and toggle functionality, implemented by naming an object's resource *MouseClickState* or *MouseClickToggle*, is also supported for backward compatibility. The value of the object's *MouseClickState* resource is set to 1 when the object is clicked on with the mouse, and to 0 when the mouse button is released. The value of the *MouseClickToggle* resource is alternated between 0 and 1 every time the object is clicked on.

The *MouseClickState* and *MouseClickToggle* resources must be visible at the proper place in the object hierarchy. If an object has *HasResources* set to *YES*, the *MouseOverState* resource will appear as a resource of the object, and the object will move every time its is clicked on with the mouse. If the object is part of a group and the group's *HasResources* is set to *YES*, but the object's *HasResources* is set to *NO*, the *MouseClickState* resource will appear as a resource of the group and the object will move every time the mouse clicks on any object in the group.

Integrated Events

Object Selection Events

Low-Level Object Selection Events

Low-level object selection events are processed automatically and do not require any setup in the drawing for individual objects. The viewport's *ProcessMouse* attribute has to include a combination of the *Click* and *Move* masks to receive selection events on MouseClick and/or MouseOver events.

At run time, the program's *Input* callback will receive an object selection message containing the mouse action that triggered the selection. The message will contain a list of all selected objects on the lowest level of the object hierarchy.

Refer to the *GLG Programming Reference Manual* for more details of the *Input* callback and custom event handling. The *Object Selection Message Object* is described on page 366 of the *GLG Programming Reference Manual*.

The *Select* callback provides a simple name-based alternative that supplies a list of names of all objects selected with the mouse click, and does not depend on the settings of the *ProcessMouse* attribute.

Custom Object Selection Events and Commands

A targeted per-object selection event can be defined by attaching an action with *ActionType=SEND_EVENT* or *SEND_COMMAND* to an object in the drawing using the *Object, Actions, Add Custom Mouse Event* or *Object, Actions, Add Command* menu options of the Enterprise Edition of the GLG Builder. The HMI Configurator also allows users to add SEND_COMMAND actions. An action defines a command or a custom event to be triggered when the object is selected with either MouseClick and MouseOver, and contains any command data needed to execute the command.

The use of integrated actions attached to objects at design time simplifies the application code that handles object selection at run time. Instead of processing a list of selected objects and relying on hard-coded object names for determining the type of action to be performed, the application code receives and processes an action message containing detailed action data. Integrated actions make it possible to process commands associated with the selected object in a generic fashion and handle arbitrary drawings created by the user. For example, one object may issue a *GoTo* command when it is selected with the mouse, while another object may issue a *PopupDialog* command.

Both mouse click and mouse over selections are supported, depending on the setting of the action's *Trigger* attribute: MOUSE_CLICK or MOUSE_OVER. An action's *ProcessArmed* attribute may be used to activate the action only when the *Control* key is held down.

To enable processing of custom mouse events, the viewport's *ProcessMouse* attribute has to include a combination of *Click* and *Move* masks depending on the desired selection type.

At run time, the custom object selection message is processed in an application's *Input* callback, which executes the action defined by the command or custom event using the data contained in the action object. The source code of the *GlgSCADAViewer* demo provides examples of handling custom events and commands in an application code.

Refer to the *GLG Programming Reference Manual* for information on using the *Input* callback for handling custom events. Refer to the *Custom Event Message Object* section and the *Command Message Object* section of the *GLG Programming Reference Manual* on page 360 for more information on the message objects.

The old-style (prior to v. 3.5) custom events, implemented by naming an object's resource *MouseClickEvent* or *MouseOverEvent*, are also supported for backward compatibility. If an object have one of these resources, an corresponding custom event will be generated when the object is selected with the mouse. The *EventLabel* of the message object will contain the value of the named resource that triggered the message.

The menu options for adding the old-style custom events are disabled by default in the Builder, but may be enabled by changing the value of the *DisablePre3-5Menus* parameter in the *glg_config* file.

If a drawing does not use the old style custom events, mouse feedback and tooltips, the *GlgDisablePre35ObjectEvents* global configuration variable described on page 179 may be used to disable them, which may decrease CPU load when moving the mouse over a large drawings.

Input Object Events

Low-Level Input Object Events

Low-level input object events are generated automatically when the user interacts with GLG input objects, such as buttons, toggles, sliders or spinners, and are not controlled by the settings of the GLG drawing.

When a low-level event is generated, the program's *Input* callback receives a message containing the event type as well as information about the input object that generated the event, and uses this information to perform appropriate actions depending on the input object and the event type that generated the message.

Refer to the *Input Objects* chapter on page 191 for more information on the GLG input objects. Refer to the *GLG Programming Reference Manual* for details on the usage of the Input callback and input object events.

Input Command Actions and Custom Events

Input actions are activated by specific input object events, such as activation of a push button or changing a value of a slider. Unlike actions that are activated by MouseOver and MouseClick events, and can be added to any object, input actions can be added only to input objects (viewports with the Input Handler), such as buttons, toggles or sliders.

An input command or custom event action can be defined by attaching an input action with *ActionType=SEND_COMMAND* or *SEND_EVENT* to an input object in the drawing using the *Object, Actions, Add Input Command* or *Object, Actions, Add Input Action* menu options of the Enterprise Edition of the GLG Builder. The HMI Configurator also allows users to add input command actions. An action defines a command or a custom event to be triggered and has parameters that control what type of user interaction triggers the action. An input action also contains data needed to execute the command.

The use of integrated actions attached to input objects at design time simplifies the application code that handles user interaction. Instead of processing low-level input events and relying on hard-coded input object names for determining the type of action to be performed, the application code receives and processes an input action message containing detailed action data. Integrated actions make it possible to process commands associated with input objects in a generic fashion and handle arbitrary drawings created by the user. For example, a button in the drawing may issue a *GoTo* command when it is clicked on, while another button may issue a *Quit* command.

The *InputAction* attribute of the input action defines the type of input activity that triggers the action. Input actions are always processed and do not depend on the settings of the viewport's *ProcessMouse* attribute.

At run time, an application's *Input* callback receives an input action message when the action is triggered. The message contains the action object together with any associated action data, which are processed by the application code. The source code of the *GlgSCADAViewer* demo provides examples of handling input actions in the application code.

Refer to the *GLG Programming Reference Manual* for information on using the *Input* callback for handling custom events. Refer to the *Custom Event Message Object* section and the *Command Message Object* section of the *GLG Programming Reference Manual* on page 360 for more information on the message objects.

Input Object Set and Reset Actions

Certain actions, such as setting or resetting the value of an attribute, do not require any application code and may be defined at design time using the Enterprise edition of the Builder. To attach this type of actions to an input object in the drawing, use the *Object, Actions, Add Input Action* menu option and set the action's *ActionType* attribute to *SET_STATE*, *RESET_STATE* or *TOGGLE_STATE*.

The SET_STATE action type sets the value of the action's State attribute to 1 when the input action is activated. The RESET_STATE action type sets the State attribute's value to 0, and TOGGLE_STATE toggles the attribute between 1 and 0 every time the action is activated. For example, a drawing may contain two buttons, a *Start* button with the SET_STATE input object action, and a *Stop* button with the RESET_STATE action. If the *State* attribute of each input action is constrained to the *Enabled* attribute of a timer transformation that drives a drawing animation, the animation may be started or stopped by using the buttons without a need to write any supporting application code.

The *InputAction* attribute of the input action defines the type of input activity that triggers the action. Input object actions are always processed and do not require any settings of the viewport's *ProcessMouse* attribute.

Custom Fonts and Font Tables

GLG applications have multiple options for customizing fonts used to render text objects in GLG drawings. A list of fonts used by the drawing is defined in a GLG *FontTable*; a font table can be customized to provide custom fonts. If a custom font table is not provided, GLG drawings use a default font table.

A custom font table can be specified in one of the following ways:

- A custom font table for a GLG drawing can be defined and stored in the drawing using *More, Add Font Table* in the viewport's *Properties* dialog.
- Multiple drawings can share a custom font table stored in an external file using the viewport's *FonttableFile* attribute.
- A custom font table stored in a file can be set as a global default using the *GlgDefaultFontTableFile* global configuration resource. All drawings that use the default font table will inherit this global default.

Refer to the description of the *FontTable* object on page 147 for more information.

Internationalization and Localization Support

Cross-Platform I18N Support

There are several features of the GLG Toolkit that support deployment of the GLG drawings in different language locales. There are two options for supporting different system locales:

- The text strings defined in the drawing may be stored in the **system locale's encoding** and rendered using fonts with locale-specific character sets. Both single-byte and multi-byte locales are supported.
- The text strings may also be stored in the **UTF-8 encoding** and rendered using fonts with UTF-8 encoding. This ensures proper string rendering regardless of the system locale and also allows mixing different character sets in the same string.

Text strings defined in the drawing are rendered using fonts defined in the drawing's font table. A default font table is used, unless the user defines a custom font table, see page 96.

An application can use a single font table for all drawings, or define different font tables for each drawing or even for each individual viewport in the drawing. Each font table can define a mix of locale-dependent and UTF-8 fonts.

Each font in the font table has attributes that define the font names to be used in different programming environments, such as Windows, X Windows and Java. This makes it possible to create drawings that may be shared between various platforms and different programming environments.

A drawing may be localized using the String Export and Import features described below, which are used to translate all text strings defined in a drawing into a different language. The drawing may be localized and saved in the Builder or localized dynamically at run time.

Displaying a localized version of the drawing may require different fonts. An application can define the drawing's font table at run-time using a viewport's *FonttableFile* attribute described on page 95; the content of the default font table may also be supplied by an external font file specified by the *GlgDefaultFontFile* global configuration resource, see page 346 of the *GLG Programming Reference Manual*.

Internationalization Support in the Java and C#/.NET Versions of the Toolkit

Java and C# use the UTF-16 version of the UNICODE for the internal representation of text strings, which makes it possible to render character sets of any locale with no additional actions. The only internationalization issue in the Java and C#/.NET version of the Toolkit is related to proper string decoding when loading drawings saved in various system locales.

The GLG Java API provides versions of the *LoadObject* and *LoadWidget* methods that have a *charset_name* parameter which specifies the charset in which the drawing was saved. The C#/.NET version of the GLG API uses a similar *encoding* parameter for these methods. When the drawing is loaded, all strings in the drawing will be decoded from the specified charset and converted to the internal UTF-16 Java string format. If the *charset_name* or *encoding* parameters are not supplied, the platform's default charset will be used.

In Java, the *GlgBean*, *GlgJBean* and *GlgJLWBean* components have the *CharsetName* property which specifies the charset for decoding drawings loaded into the beans on the per-bean basis. The *GlgDefaultCharset* global configuration resource may also be used to set the default GLG charset different from the default system charset for the whole application.

In C#/.NET, the *GlgControl* component provides the *GetEncoding* and *SetEncoding* methods for setting default encoding on per-control basis.

If a string in a drawing has the *UTF8Encoding* flag set to YES, the string is automatically decoded from the UTF-8 encoding to the internal UTF-16 representation regardless of the *charset_name* parameter of the load method.

String Encoding in the GLG Drawings

All strings stored in S data **objects** (such as the *TextString* attribute of a text object, tooltip or custom property of S type) may be stored in either the encoding of the system locale or the UTF-8 encoding, depending on the setting of their *UTF8Encoding* flag.

All other strings that are **not objects**, such as object names, tag names or tag sources, are stored in the encoding of the current system locale.

Multi-Byte Character Set Support

The GLG font objects defined in the viewport's font table support various character sets, both single and multi-byte. Each font has the *MultiByteFlag* attribute that determines the type of the font: SINGLE_BYTE, MULTI_BYTE or UTF8.

On Windows, each font also has the *FontCharset* attribute that defines the version of the font to use. If it is set to DEFAULT_CHARSET, the version of the font with the charset of the current system locale will be used, otherwise the version of the font with the specified charset will be used. If *MultiByteFlag* is set to UTF8, the UNICODE version of the font will be used and the value of the DEFAULT_CHARSET attribute is ignored.

In the X Windows environment, the font's encoding is specified as a part of the font's name, and the *MultiByteFlag* attribute determines how to handle the font name. If it is set to SINGLE_BYTE, the font's *XFontName* attribute is handled as a single font name, otherwise it is handled as a comma-separated font set containing one or more fonts.

For all fonts of the default font table, as well as fonts with the *FontCharset* attribute set to DEFAULT_CHARSET on Windows, the actual value of *MultiByteFlag* is determined automatically based on the system locale. The *GlgMultibyteFlag* global configuration variable may be used to specify the value of *MultiByteFlag* that overrides the automatic setting for these fonts.

UNICODE and UTF-8 Support

The Unicode is supported via the UTF-8 character encoding, which, due to its ASCII compatibility, is the version used the most across different platforms and on the web. The unicode support provides a locale-independent way to render text, and also allows to mix characters from different language locales in one text string.

There are two parts related to UTF-8 support:

- **Encoding used to store the string**
  All strings stored in S data objects (such as the *TextString* attribute of a text object, tooltip or custom property of S type) may be stored in either the encoding of the system locale or the UTF-8 encoding. Any string may be stored in the UTF-8 encoding even if it is rendered with the non-UTF8 font, in which case an automatic conversion will be performed at the rendering time.
- **Font used to render the string**
  Any string (UTF8 or non-UTF8) may be rendered with either UTF8 or non-UTF8 font. If the string's encoding does not match the encoding of the font, an automatic conversion will be automatically performed. If the string contains characters that are not present in the font, they are replaced with a default character.

To enable this feature in the X Windows environment, the font objects and the text rendering drivers support the use of both the individual fonts as well as the font sets. On Windows, fonts with *MultiByteFlag* set to UTF8 use the UNICODE version of the font, performing an automatic conversion to the wide character representation and rendering using the wide character version of the text drawing functions.

Using UTF-8 Locale on Linux/Unix

In the Linux/Unix environment, the UTF8-based system locale, such as *en_US.UTF-8* may be used.

The fonts in the default font table as well as the default Motif fonts (used by the native widgets in the drawing and by the Motif controls in the GLG Editor) support the ISO Latin1 extended ASCII character set (ISO 8859-1). The *MultiByteFlag* attribute of all fonts in the default font table will be automatically set to UTF8 when the UTF8-based system locale is detected.

A custom font table can be specified to render different character sets in the GLG drawing. The font table is defined as an attribute of the drawing's viewport. The fonts in the font table must use ISO-8859 or some other encoding supported by the UTF8 font sets. The *XFontName* attribute of each font may list several comma-separated font names that will be handled as a **font set** for rendering UTF-8 characters. The font's *MultiByteFlag* must be set to UTF-8.

In order to use different character sets in the GLG Editor's text fields, buttons and toggles, as well as in the native controls in the drawing, an appropriate font with the ISO10646 encoding have to be specified via the Motif's *renderTable* resource. Here is an simple example of X defaults for Motif fonts:

```
*renderTable: rt1
*rt1.fontType: FONT_IS_FONTSET
*rt1.fontName: -*-*-*-*-*--13-*-*-*-*-*-iso10646-1
*Text.renderTable: rt1
*TextField.renderTable: rt1
*PushButton.renderTable: rt1
*ToggleButton.renderTable: rt1
*OptionMenu.renderTable: rt1
```

Due to the Motif's use of the UTF-16 version of UNICODE for the internal representation of strings, it needs fonts with ISO10646 encoding to support the UTF8-based locale.

Cross-Platform Use Note: Windows has the UTF-8 codepage but does not provide the UTF-8 system locale. Creating a drawing using the UTF-8 system locale on Linux/Unix can create problems when deploying the drawing on Windows. To avoid portability problems with drawings created in the UTF-8 system locale, set the *UTF8Encoding* flag to YES for all string attributes that use non-ASCII characters and use ASCII characters for strings that are not objects (such as object names, tag name and tag sources). The *gconvert* utility's *-set-utf8* command-line option may be used to set the *UTF8Encoding* flag of all string objects in the drawing; see page 326.

In some linux distributions the "locale -a" command lists UTF-8 locales as "utf8". When setting UTF-8 locale, make sure that "UTF-8" is spelled exactly this way, all caps and with a dash, otherwise Motif will not recognize the UTF-8 locale. For example, set "*LANG=en_US.UTF-8"* for English UTF-8 locale.

Localization Support

The String Export and Import features provide a way to translate all text strings defined in the drawing into a different language, either at a design time using the Builder or at run-time, using the GLG API functions to import the string translation file.

The String Export feature is used to export all strings defined in the drawing into an ASCII string translation file. The string translation file contains an entry for each exported string and may be edited using a text editor. Each string entry contains the name of a string resource which helps identify how the string is used, and two copies of the string. Each item in the string entry is separated by two separator characters. The name of the string resource and the first copy of the string are used to identify the string and should not be changed.

When the file is translated, the second copy of the string may be replaced with a new string representing the text in the local language and local character set. The String Import feature is then used to load the translated file and replace strings in the drawing with the new translated strings from the string import file.

The strings in the exported translation file are separated with a two-character separator. Two double quotation characters are used as the default separators, but that can be changed by defining the GLG_STRING_SEPARATOR environment variable to supply a two character string to be used as a separator. In the GLG API, separator characters are supplied as function parameters.

The following is an example of the string conversion file using the default *""* separator characters:

```
HEADER ""GlgStringConverter""2""

# Comment: Translated Strings
""Resource1""String1""Stroka1""
""Resource2""Label2""Metka2""
""Resource3""Multi-line
Text""Tekst b dve
stroki""
```

The first line of the file contains version information and should not be modified. Any characters between the string's terminating separator and the separator at the beginning of the next string are ignored, allowing for comments and blank lines to annotate the string file. A string may extend to several lines, as shown in the last conversion file entry.

If some strings should be not translated, their entries may be left unchanged, or even better, removed from the file. Only the strings defined in the translation file will be replaced.

Builder Features

The *File* menu of the Graphics Builder contains *Export Strings* and *Import Strings* options to export or import strings of the drawing loaded in the Builder. When editing focus is in a viewport, only that viewport's strings are exported or imported, providing a way to export or import strings for only a part of the loaded drawing. After the exported string file has been translated, it can be imported back and the new localized drawing may be saved. This allows the translator to edit the strings in a text file instead of finding all text strings in the drawing.

Drawing Conversion Utility Options

The drawing conversion utility supports *"-export_strings <filename>"* and *"-import_strings <filename>"* options for exporting and importing the strings in a batch mode.

GLG API Methods for Run-Time Localization

The *GlgExportStrings* and *GlgImportStrings* methods are provided in both C/C++, ActiveX, Java and C#/.NET versions of the Toolkit's API. The application may utilize a single drawing and provide translation files for multiple locales to localize the application at run time. The run-time localization is performed by loading an appropriate translation file depending on the system's locale into the drawing using the *GlgImportStrings* method.

Using String Import Feature And Unicode With the Java Version of the Toolkit

The Java version of GLG API supports Unicode and uses InputStreamReader to read native characters from the strings translation file, decoding the characters using character set defined by the current locale. The *ImportStrings* method of the GLG Java API has an encode parameter, allowing an application to load string translation files with encoding different from the current locale setting.

It is common in Java environment to use ASCII encoding with *\u* Unicode escape character for representing non-ASCII characters. Such encoding, however, does not correspond to any system locale, is not supported by InputStreamReader and must be converted to some supported encoding before processing. For example, in Java source code and resource bundles, such conversion is done at the compilation time by the Java compiler, which converts the *\u*-encoded ASCII to UTF-16 characters in the generated Java byte-code.

For string translation files, an equivalent conversion has to be performed. Java provides a *native2ascii* utility that can be used to convert from Unicode to ASCII encoding with *\u* Unicode escape, or vise versa. The following command line shows how to run the utility to convert the ASCII *strings.txt* file to UTF-8 unicode, which can be used with any UTF-8 locale:

```
native2ascii -reverse -encoding UTF8 strings.txt strings2.txt
```

The converted strings2.txt file may then be used with the Java program. To convert from Unicode back to ASCII with *\u* encoding, run the *native2ascii* utility without the *-reverse* option.

Data Connectivity Features

Resource-Based Data Access

All objects and object attributes are inherently dynamic and may be accessed and changed at run-time by using the API's *SetResource* methods. Not only the properties such as colors or line width, but also resources that define rotation angles, move distances, scale factors, thresholds and many others may be set from a program via programming API by specifying the resource name and a new value, as shown in the following example:

```
GlgSetDResource( drawing, "Meter1/Value", 25. );
GlgSetSResource( drawing, "Meter1/Label", "MPH" );
GlgSetDResource( drawing, "Group1/Polygon1/LineWidth", 3. );
GlgSetDResource( drawing, "Group1/Polygon2/RotateAngle", 30. );
GlgUpdate( drawing );
```

The GlgUpdate method is invoked at the end to update and render the drawing with the new resource values.

The Toolkit's resource-based access to data provides a unified and compact programming API which is ideally suited for applications that need an access to all objects defined in the drawing and complete control over the objects' attributes.

Tag-Based Data Access and Database Connectivity

Tags provide an alternative way to access dynamic attributes defined in the drawing. A data tag may be attached to any dynamic parameter or object attribute to define data connectivity. It may also be used to store user-defined information associated with the attribute. The tag's *TagSource* attribute provides a way to map resources of the drawing to database fields, which is commonly used for data connectivity in process control applications.

While the resources are hierarchical and require the application to know the exact resource path of each resource, the tags are global and are accessed in an application as a flat list. This enables the application to query and use the tags defined in the drawing without the need to know the exact structure of the drawing.

The tags may be assigned to object attributes in the Builder. The Builder also provides a Tag Browser for browsing and editing tags defined in the drawing. With the Tag Browser, a user can edit each tag by assigning a database field to its *TagSource*. A custom Data Browser may be integrated with the Builder to allow the user to browse and select a *TagSource* from a list of available database tags. The *TagName* and *TagComment* attributes of the tag object help the user identify the tag in the Tag Browser or attach custom information to the attributes.

Using Tags as Global Resources

At run-time, an application may use tags as global resources. Attaching tags to important resources of the drawing allows the user to easily browse them in the Builder. By assigning meaningful names to the *TagSource* attributes, the application can access the resources by their associated *TagSource* as shown in the following example:

```
GlgSetDTag( drawing, "PressureMeterValue", 25. );
GlgSetDTag( drawing, "PressureAlarmState", 0 );
```

Here the tags are used in virtually the same way as resources, with the *TagSource* being handled as a global resource name. Unlike resources, *TagSources* do not have to be unique. If several tags in the drawing have the same *TagSource*, invoking *GlgSetTag* with this *TagSource* will set values of all the tags. This may be interpreted as a programmatic way of handling constrained values without actually constraining objects in the drawing. The application may also easily query all tags defined in the drawing using the *GlgCreateTagList* method.

Using Tags for Database Connectivity

Tags may also be used in a more sophisticated way to store database connectivity information in the drawing by attaching a tag to each resource in the drawing that needs to be updated from a database. The tag's *TagName* attribute is set to a logical name that helps to identify the tag while browsing, and the *TagSource* is set to the name of the database field that will provide the data for the resource the tag is attached to. A user can browse all tags defined in the drawing and edit their database data sources by changing the tags' *TagSource* attributes.

The *TagAccessType* and *TagEnabled* attributes provide additional control over the use of a tag in an application. Refer to the *Tag* section on page 137 for more information.

The *Tag Export* and *Tag Import* features described in the next section may also be used for editing or remapping all tags in the drawing, either at design time in the Builder or at run-time in an application.

At run-time, a process-control application may load the drawing, query the list of tags defined in the drawing with the *GlgCreateTagList* method and subscribe for updates for the process database fields defined by the tags' *TagSource* attributes. When data changes, the application may set the new data values by invoking the *GlgSetTag* method, passing the *TagSource* and the new data value for each tag as shown in the source code of Tags Example, which is provided for both C/C++. Java and C#/.NET versions of the Toolkit.

Tag Export and Import Features for Run-Time Tag Mapping

All tags defined in the drawing may be exported into an ASCII tag file for editing them via a batch script or a text editor. The modified tag file may then be imported back into the drawing, importing the modified *TagName* and *TagSource* attributes.

The tag export and import features provide a way to modify the tags defined in the drawing to map them to the database fields of a specific process database. It may be done either at design time using the Builder, or at run-time using the GLG API functions to import the tag file.

The tag export feature exports all tags defined in the drawing into a tag file. Each tag entry contains two copies of the tag's *TagName* and *TagSource* attributes separated by two separator characters. The first copy of the *TagName* and *TagSource* attributes is used to identify the tag and should not be changed. The second copy of the attributes may be changed to modify the tag. The new value of the *TagSource* attribute supplies the database field associated with the tag. The *TagName* attribute may be changed to modify the tag name in the drawing.

The import feature is then used to load the translated file and replace tags in the drawing with the new tags from the tag import file.

The strings in the exported tag file are separated with a two-character separator. Two double quotation characters are used as the default separators, but that can be changed by defining the GLG_STRING_SEPARATOR environment variable to supply a two character string to be used as a separator. In the GLG API, separator characters are supplied as function parameters.

The following is an example of the tag conversion file using the default *""* separator characters:

```
HEADER ""GlgTagConverter""2""

# Comment: Converted Tags
""AlarmState""undefined""""AlarmState""Plant3:Tank2:Alarm""
""Pressure""undefined""""Pressure""Plant3:Tank2:Pressure""
```

The first line of the file contains version information and should not be modified. Any characters between the string's terminating separator and the separator at the beginning of the next string are ignored, allowing for comments and blank lines to annotate the tag file. The example assigns new values to the *TagSource* attributes of two tags with the *AlarmState* and *Pressure* tag names, changing their tag sources from the *"undefined"* string to *"Plant3:Tank2:Alarm"* and *"Plant3:Tank2:Pressure"* respectively. The tags' *TagName* attributes remain unchanged.

If some tags should be not translated, their entries may be left unchanged, or even better, removed from the file. Only the tags defined in the translation file will be replaced.

Builder Features

The *File* menu of the Graphics Builder contains *Export Tags* and *Import Tags* options to export or import all tags defined in the drawing. When editing focus is inside a viewport, only that viewport's tags are exported or imported, providing a way to export or import tags for only a part of the loaded drawing. After the file has been modified, it can be imported back and the new drawing may be saved.

Drawing Conversion Utility Options

The drawing conversion utility *gconvert* supports *"-export_tags <filename>"* and *"-import_tags <filename>"* options for exporting and importing the tags in batch mode.

GLG API Methods and Run-Time Tag Mapping

The *GlgExportTags* and *GlgImportTags* methods are provided in both C/C++, ActiveX, Java and C#/.NET versions of the Toolkit's API. The *GlgImportTags* method may be used to load database connectivity information from an external tag file at run-time. An application may utilize the same drawing with different databases, using the tag file to map the tags in the drawing to the database fields.

The *GlgCreateTagList* API method may be used in an application to query all tags defined in the drawing and obtain database mapping from the tag's *TagSource* attribute, as shown in the Tags Example source code.

Custom Properties for Storing Application-Specific Data

Custom Properties may be attached to any graphical object to store application-specific data related to the object. The custom properties are attached using the *Object, Custom Properties, Add Custom Property* menu and are saved with the drawing. The object's custom properties can be retrieved in a program as object's resources, using their resource names.

Custom properties can be organized hierarchically using lists. The *Add Custom Property* menu has options for adding custom properties of different data types as well as lists of custom properties. Lists may be used to group large collections of custom properties for easier access.

Integrated Alarms for Value Monitoring

Alarms may be attached to any data object to monitor its value. For the D (double) data objects, several RANGE alarm types are available to detect when the value goes out of an application-defined range. Both the *High-Low* and *HighHigh-LowLow* ranges are supported. For all data types a CHANGE alarm may be used to detect a value change.

When alarm is activated, an alarm message is generated. The message contains a user-defined alarm label as well as other related information that may be used by an application to process the alarm. For the range alarms, a message is generated when a value is going out of range, as well as when it falls back into the range, making it possible for the application to set and reset an alarm display.

An application can install an *alarm handler* to process alarms. The alarm handler is global and processes all alarms for the entire application. The handler can display alarms in a scrolling alarm list or perform any other application-specific alarm handling.

GLG alarms are "graphical alarms" intended to be used with the graphical displays to visualize current alarm conditions in a drawing. The alarms are associated with drawings and are active only for the drawings that are loaded and displayed. To monitor all alarms in the process database regardless of the drawing being displayed, a different alarm service has to be used that monitors the data regardless of the graphics.

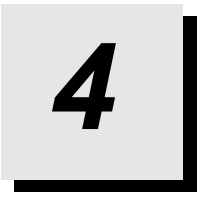
Public Properties for Creating OEM Components

The Enterprise Edition of the Graphics Builder provides a feature that allows to designate some properties of GLG objects as public properties. When an object with public properties is used in the HMI Configurator, the *Public Properties* dialog displays only the properties of the object that are marked as public.

This feature is used by system integrators to create custom components with a predefined set of properties to be edited by the user. Using public properties, the components can expose only certain properties of a component for easy editing, while hiding the rest of internal properties to protect them from being changed.

Chapter 4

GLG Objects



In addition to being familiar with the overall structure of a GLG drawing, it is useful to know about the variety of objects you are liable to see in such a drawing. This chapter contains a description of most of the objects that make up a drawing.

Note that there are some objects that are not described here. Some of these are never seen by the user, and others are seen, but may not be available to the user. The objects can be easily divided into the graphical and the non-graphical objects.

The graphical objects are those objects that are readily visible to a user looking at a GLG drawing. They may be divided into two groups, the simple and the advanced objects. The simple objects are as follows:

Polygon
The basic graphical object, used for lines and shapes.

Parallelogram
Another sub-class of polygon, for parallelograms and rectangles.

Rounded Rectangle
A rectangle with rounded corners, a sub-class of polygon.

Arc
Represents shapes with round edges, such as arcs and circles. A sub-class of polygon.

Ellipse
An ellipse, rendered as a special case of a rounded rectangle.

Spline
A multi-point Bezier or Catmull Rom cubic spline, used to render free shape curves.

Text
For placing text in a drawing.

Marker
A small design for marking a point position.

Image
For placing GIF, JPEG, PNG and BMP images in the drawing.

GIS Object
An integrated map object for embedding images generated by the GLG Map Server into a GLG drawing and automatically handling Map Server requests and user interaction.

Viewport
This is the drawing surface for a GLG drawing, and a container for other objects. This object controls the view a user has of all the objects within it.

Screen
Controls aspects of a viewport's appearance.

You can use the simple graphical objects to create a wide variety of drawings that include elaborate animations. However, the simple objects do not use many of the advanced features of the GLG drawing structure. The advanced objects allow a user to define complex collections of simple

objects, which can then be manipulated as if they were simple objects. This ability to add layers of complexity to a drawing is the key to the power of a GLG drawing. The list below shows the advanced graphical objects:

Group
Used to assemble a collection of simple objects into a complex one. The groups may be used to hold collections of graphical or non-graphical objects.

Connector
A recta-linear or arc path for connecting objects.

Reference
A *Reference* object is a wrapper around a group of objects used as a "template". The *Reference* object may be used to replicate the same template in multiple places in one drawing or in multiple drawings. It may also be used as a convenient wrapper for positioning a group of objects using a single anchor point. There are three flavors of the *Reference* object:

- ***Container***
  Encapsulates a collection of other objects and provides a single control point for positioning it in the drawing. Each container has its own independent copy of the template.

- ***SubDrawing***
  Replicates copies of a template object in a drawing. When the template is changed, all subdrawings that use the template will change as well. The template may be included in the same drawing or stored in a separate drawing file. Subdrawing dynamics may be used to alter the SubDrawing's appearance at runtime.

- ***SubWindow***
  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. The template must be a viewport object containing the drawing to be displayed.

  The *SubWindow* may also be used as a subdrawing for interface objects that require two control points, such as buttons or menus. Using a subdrawing for these objects is not convenient, as a subdrawing has only one control point, while it is easier to define a button's position using two points. Using a subwindow for buttons allows to change the button appearance in all drawings by editing a single template.

Series
A one-dimensional collection of objects defined by a template object, a number of repetitions, and a path on which to repeat them. The path need not be a straight line.

Square Series
A two-dimensional collection of objects defined by a template object, a number of repetitions, and a grid on which to repeat them.

Polyline
A collection of points and connecting lines.

Polysurface
A two-dimensional collection of points and connecting faces.

Frame
Provides an array of control points, to which other objects may be constrained.

Chart objects are specialized objects used to render real-time charts and their subobjects:

Chart
A specialized object used to render real-time charts.

Plot
Represents individual plot lines in a chart.

Level Line
Displays a horizontal line representing a threshold value in a chart.

Axis
A specialized object for rendering axes in a chart as well as stand-alone ruler objects.

Legend
A specialized object for displaying chart legends.

GLG non-graphical objects are used to control the appearance and behavior of graphical objects. They are as follows:

Data
Used to hold a data value.

Tag
May be attached to a data object to mark it as a global resource or to define database connectivity for its data value.

Attribute
Similar to the data object, but used to hold an attribute value.

History
Used to control scrolling behavior in graphs.

Alias
Specifies an alternative, application-defined (logical) name for accessing arbitrary resource hierarchies.

Color Table
Controls the selection of colors available for display in a drawing

Font Table
Controls the selection of fonts available for use in a drawing.

Font
Specifies the font to use for a text object.

Rendering
Specifies an extended set of rendering attributes.

Box Attributes
Specifies attributes of a box drawn around the text object.

Line Attributes
Specifies rendering attributes of plots, grid and level lines, cross-hair cursors and backgrounds of chart objects, as well as tick attributes of axis objects.

The last section of this chapter describes the variety of available transformations used to implement dynamic behavior, as well as the alarm objects used to monitor data values. Though these are in the same category as the other non-graphical objects, they are complex enough that they merit their own section.

Transformation
Describes a transformation associated with the object.

Alarm
Describes an alarm associated with the data or attribute object; it is implemented as a special type of a transformation object.

The objects and their attributes are described in greater detail in the rest of this chapter.

NOTE: All attributes of GLG objects are stored as **S** (string), **G** (geometrical), or **D** (double-precision scalar value) data values. Where an attribute is described below as having a limited number of values, it is actually stored as a scalar (D) value, often as the index into an array of possible values.

Common Attributes

All GLG objects share a basic structure. That is, they are all sub-classes of the basic object class. This means that most objects share a few basic attribute types. Most of the common attributes are displayed in the separate area at the top of the respective property dialogs in the Graphics Builder. Some of the most commonly encountered types are described in the list below.

Common Attributes

Name
An object's name is its entry in the resource hierarchy. An object need not have a name.

HasResources
Controls where in the resource hierarchy the names of an object's children appear. For more information about this flag and the subject of "resource transparency", see the *Hierarchy of Resources* section in *Structure of a GLG Drawing*.

Xform
A slot for attaching an optional *transformation* object. A *Concatenate* transformation type may be used to attach a list of several transformations. The list is accessed by using the *Edit Dynamics* buttons in the Builder. The Xform attribute assumes the value of NULL if no transformation is attached.

Common Attributes of Graphical Objects

Visibility
The *Visibility* attribute of a graphical object controls whether the object is displayed or not. This attribute is a floating point value that can range between 0 and 1. A value of 1 indicates that the object is visible; a value of 0 makes the object invisible. The values between 0 and 1 make the object transparent, with the object transparency increasing as the visibility value decreases. The transparency effect does not apply to the viewports.

Negative values of *Visibility* have a special meaning and are handled as dimming. A visibility value between -1 and 0 causes object colors to be dimmed by decreasing their

saturation. The closer the value is to 0, the duller the object colors become. If the visibility value is negative but its absolute value is bigger than 1, the object's colors will be brightened. The dimming effect may be used to change appearance of icons when they are desensitized.

The dimming and transparency of a container object are inherited by all its children. The child's visibility may further alter the child's rendering, with the effective visibility value calculated by multiplying the child's visibility value by the visibility values of all its parents.

For environments with the OpenGL support (and also Java and C#/.NET versions of the Toolkit), the transparency is rendered as true alpha-blending. For the GDI versions of the Toolkit, the transparency is supported only in the Unix environment, where it is simulated using dithering patterns, except for the image objects.

Transparency is not supported in the Postscript output.

MoveMode

This attribute specifies how the object's control points are modified when the object is moved with the mouse and may have the following values:

GLG_MOVE_POINTS

Moves all controls points of the object, but does not move control points of geometrical transformations attached to the object, if any. For example, if the object has a rotate transformation attached, the center of rotation will not be moved with the object when the object is repositioned.

GLG_STICKY_CENTER_MODE

Moves all control points of the object as well as the control points of geometrical transformations attached to the object, if any. This setting may be used when the object has to rotate around its center: when the object is moved, the center of rotation will moved with the object to preserve its position relatively to the object. This is the default for newly created objects.

GLG_MOVE_BY_XFORM

Instead of moving the object by changing the spatial coordinates of its control points, the object is moved by creating a matrix move transformation and attaching it to the object. The transformation moves the object without reassigning coordinates of its control points, which is useful if it is needed to preserve the original coordinates of an object's points.

In addition to moving an object with the mouse, the attribute also controls how the object is modified using the Builder's *Transform Object Points* button or via the GLG API functions which move or transform an object.

CoordFlag

This advanced attribute specifies the coordinate system used for rendering the object. If set to GLG_INHERIT_COORD_SYSTEM ("INHERIT" label), the coordinate system used to interpret coordinates of the object's control points is inherited from the viewport in which the object is drawn. When *CoordFlag* is set to GLG_ABS_SCREEN_COORD_SYSTEM ("ABS_SCR" label), the screen coordinate system is used. The GLG_ABS_FLIPPED_SCREEN_COORD_SYSTEM setting ("GLG_SCR" label) uses the GLG screen coordinate system which has the same upper left origin as the screen coordinate system, but has the Y axis pointing up. The screen coordinate systems may be used for rendering prompts and overlays which do not change their position when the drawing is zoomed or resized.

History

A slot for attaching one or more optional *History* objects. A group is used to attach a list of history objects. The list is accessible by using the *Edit History* buttons on the *Object* Menu. The attribute assumes the value of NULL if no *History* objects are attached.

Aliases

A slot for attaching one or more optional *Alias* objects. A group is used to attach a list of aliases. The list is accessible by using the *Edit Aliases* buttons on the *Object* Menu. The attribute assumes the value of NULL if no *Alias* objects are attached.

CustomData

A slot for attaching one or more optional *Custom Data Properties*. A group is used to attach a list of properties. The list is accessible by using the *Edit Custom Properties* buttons on the *Object* Menu. The attribute assumes the value of NULL if no *Custom Properties* are attached. In order to minimize memory consumption, custom properties may only be attached to graphical objects (polygons, viewports, text objects, etc.).

Rendering

A slot for attaching an optional *Rendering* object to control an expanded set of rendering attributes, such as gradient fill, cast shadows, fill level and arrow heads. The rendering object is accessible by using the *Add/Edit Rendering* buttons in the *Object Properties* dialog. To delete the rendering object, use the *Delete Rendering* button in the Rendering Object properties. The attribute assumes the value of NULL if no rendering object is attached. See *the Rendering* section on page 140 for details.

When a rendering object is added to all objects in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added rendering objects (the attributes are constrained if the default *Constrained Clone* setting is used).

Common Attributes of Attribute and Data Objects

Global

This attribute is used to establish the relationship between the attributes of an object, and the attributes of copies of that object. In general, the global attributes of two copied objects are constrained to each other. For example, if a polygon has a global *FillColor* attribute, any copies made of it using a copy operation other than a Full Copy will have their *FillColor* attributes constrained to that of the original. The *Global* attribute has three possible values for controlling such attribute constraints: GLOBAL, LOCAL, and SEMI-GLOBAL. The last value is used by specialized copy operations; see, for example, *GlgCloneObject* method on page 137 of the GLG Programming Reference Manual.

The Global attribute also has two special values: BOUND and NONE. The BOUND value is used for rebinding attributes or reference objects (subdrawings and subwindows); refer to the *Bindings* section on page 108 for more information. NONE is used for some special attributes to prevent them from ever being constrained; refer to the *GlgCloneObject* method on page 137 of the GLG Programming Reference Manual for more details.

Though most of the objects described in this chapter use these attributes, they are not separately described in the lists that follow.

The control points of objects are also omitted. Objects may have either a fixed (marker, text, etc.) or variable (polygon) number of control points. For objects with a fixed number of points, the points can be accessed using the *Point1, Point2, ..., PointN* default attribute names. For objects with a variable number of points, the points can be accessed using functions which access a container's elements. A user can also make control points into named resources by granting them names in the Builder.

The lists in the following sections include only attributes with default names other then control points.

Simple Graphical Objects

Simple GLG graphical objects can be directly viewed in a drawing. These are the simplest building blocks of a picture, and most of them will seem familiar to you. The following sections introduce the GLG objects in greater depth, and provide lists of the most commonly accessed object attributes.

Polygon

The **polygon** is a basic graphical object, and is used to represent both lines and polygons. A line in a GLG drawing is simply an open polygon, while most shapes are represented by closed polygons of one sort or another. A straight line is a two-point polygon.

The polygon is basic to the structure of a GLG drawing in other ways. Arcs, rectangles, parallelograms, splines and connectors inherit their attributes from a polygon.

A simple polygon has a control point at each vertex. It also has the following attributes:

FillColor
The color of the polygon's interior, if it is filled.

EdgeColor
The color of the polygon's defining line or edge.

LineWidth
The thickness of a polygon's edge. Lines with odd line width use round line ends. Lines with an even line width use square ends. The intermediate connections of a multi-line polygon always use rounded connections.
If the OpenGL renderer is used, the maximum line width is limited by the graphics card's hardware and is 10 for most of the modern graphics cards.

LineType
The line pattern (solid, dashed, etc.) for rendering the edge of the polygon. GLG provides 32 predefined lines types shown in the Builder's Line Type palette.
OpenGL Note: The line type rendering is consistent between the GDI version of Builder, C/C++ library, ActiveX Control, Java and C#/.NET. However, rendering of some line types in viewports with OpenGL enabled in the OpenGL version of the Builder, C/C++ library and ActiveX control may differ due to the differences in the way line types are defined in OpenGL.
"**Moving Ants" Dynamics Note:** If *LineType* is greater than 32, the reminder of division of *LineType* by 32 is used as a line type, and the result of the division is used as a line type pattern offset in pixels. The length of the pattern is 32 for the GDI renderer, Java and C#/.NET, and 16 for the OpenGL renderer.
The line type pattern offset may be used for a "moving ants" animation of the line type pattern, as seen in the GLG Process Control Demo. The effect is achieved by repeatedly increasing the *LineType* value by 32, which causes the line type pattern to shift by one pixel. To avoid overflow, the *LineType* has to be periodically reset back to the initial line type value. Since the length of the pattern is 24 for the GDI renderer and 32 for OpenGL, resetting it after every 16 * 24 = 384 iterations makes it work regardless of the used renderer.
A predefined ***Flow* dynamics** may be attached to the *LineType* attribute for the line type pattern animation, see page 174.

FillType
Defines how a polygon is rendered. The choices are formed by combining the three possible choices by ORing together their binary constants:
GLG_FILL - enables rendering of the polygon's fill using *FillColor*

GLG_EDGE - enables rendering of the polygon's edge using *EdgeColor*

GLG_LINE_FILL - used with the GLG_EDGE to render the outer edges of thick lines using *EdgeColor* and the middle part using *FillColor*.

OpenType

Defines whether or not the line connecting the first and the last points of the polygon is drawn. It does not have any effect on polygons with fewer than three points.

Shading

Controls shading for an individual polygon. Shading is enabled for the viewport if it has a *Light* object added to it. The attribute may have the following values:

GLG_NO_SHADING - disables polygon shading

GLG_FILL_SHADING - enables shading of the polygon fill only (default)

GLG_FILL_EDGE_SHADING - enables shading of both the polygon's fill and edges.

AntiAliasing

Controls antialiasing of the polygon's edges and may have the following values:

GLG_ANTI_ALIASING_INHERIT
: Inherit antialiasing settings from the global setting, which is GLG_ANTI_ALIASING_INT by default. The default can be changed globally by using the *GlgAntiAliasing* global configuration resource.

GLG_ANTI_ALIASING_OFF
: Disable antialiasing.

GLG_ANTI_ALIASING_INT
: Enable antialiasing and map vertices to integer pixel boundaries. This matches the coordinate mapping of the native non-OpenGL renderer and makes the straight lines look sharper.

GLG_ANTI_ALIASING_DBL
: Enables antialiasing and uses double vertex coordinates. This setting makes the curved lines look better. It is used as a default setting for arcs, splines, rounded rectangles and plot lines of a real-time chart.

For the GDI driver in C/C++/ActiveX, the antialiasing is enabled only for the viewports that use the OpenGL renderer and have their *OpenGLHint=ON*. For Java, the attribute is ignored and the anti-aliasing is controlled globally by the *GlgAntiAliasing* global configuration resource. The attribute is always used in C#/.NET.

Point List

A list of polygon's control point. Allows you to change the order of points in the polygon as well as add and delete points from the polygon.

Rendering

A slot for attaching an optional *Rendering* object to control an expanded set of rendering attributes, such as gradient fill, cast shadows, fill level and arrow heads. The rendering object is accessible by using the *Add/Edit Rendering* buttons in the Object Properties dialog. To delete the rendering object, use the *Delete Rendering* button in the Rendering Object properties. The attribute assumes the value of NULL if no rendering object is attached. See *the Rendering* section on page 140 for details.

When a rendering object is added to all objects in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added rendering objects (the attributes are constrained if the default *Constrained Clone* setting is used).

Parallelogram

A parallelogram is a four sided polygon that includes implicit constraints to keep the opposite sides parallel. The parallelogram has only three control points and one constrained point. The attributes of a parallelogram are the same as the attributes of a polygon except that, because a parallelogram is always closed, the *OpenType* attribute is not present.

When a parallelogram is "exploded" in the Builder it becomes a regular four-sided polygon. The constraints that keep its sides parallel are removed, and the fourth point becomes a simple control point.

Rectangle

A rectangle is a special case of the parallelogram object in which each pair of adjacent sides is perpendicular. A rectangle has two control points located at the end of the rectangle's diagonal and two constrained point at the end of another diagonal which are managed automatically.

Rounded Rectangle and Ellipse

An object of type ROUNDED is used to render both rectangles with rounded corners and ellipses. Same as a parallelogram, the **rounded** object is defined by three control points, but it is initially created in the Builder as a rectangle defined by two points.

Like the parallelogram, the rounded object is simply a sub-class of the polygon, so it has the usual polygon attributes, like *LineType* and *FillColor*. It also has the following attributes that control the object's rounded corners:

Radius1
Controls an extent of rounded corners along the side of the rounded rectangle defined by the first and second control point (Y dimension when originally created in the Builder).

Radius2
Controls an extent of rounded corners along the side of the rounded rectangle defined by the second and third control point (X dimension when originally created in the Builder). If set to -1, the value of *Radius1* is used, and the size of rounded corners may be controlled with a single parameter - *Radius1*.

UnitType
Specifies units used for *Radius1* and *Radius2* attributes. The following options are available:

GLG_SCREEN_UNITS - corner radiuses are defined in screen coordinates; the size of the rounded corners stays constant.

GLG_WORLD_UNITS - corner radiuses are defined in world coordinates; the size of the rounded corners changes proportionally when the drawing is resized, but stays constant when the object is resized with the mouse.

GLG_RELATIVE_UNITS - corner radiuses are defined as coefficients in the [0;1] range relative to the extent of the rectangle's corresponding side. The size of the rounded corners changes proportionally when the drawing or the object is resized, maintaining a constant ratio between the size of the rounded corners and the length of the object's side in the corresponding direction. .

Resolution
The number of line segments used to render rounded corners. The default value for this is 7 for rounded rectangles (25 for ellipses). A larger value may be used for nicer rendering of rectangles with large corner radiuses.

If *Radius1* and *Radius2* are set to 1 in relative units, the object will render an **ellipse.** If *UnitType* is set to relative and *Raduis2* is set to -1, the rounded corners will take the whole height of the object and an equal amount of space in the horizontal direction. This results in the object being drawn as a rectangle with a round left and right side, except when the object is too small in the horizontal direction.

Arc

The Arc object is used to represent both arcs and circles. One part of an arc is a section of a circle's perimeter. A **chord** arc simply joins the two ends of the curve with a straight line, while a **sector** arc is shaped like a piece of a pie, with two straight lines joined at the center of the circle describing the extent of the third, curved side. A circle is simply the special case of an arc whose interior angle is 360°.

An arc has two control points: a center and a vector point. A vector from the center point to the vector point defines a line perpendicular to the arc plane. As you move either of the two ends of this vector, you can see the arc twist in space. The length of this normal vector is not used, only its orientation in space.

Since the arc vector is perpendicular to the arc plane, the vector point of the arc coincides with its arc's center point in the main projection. Also, visually, the vector point is on top of the center point, so selecting the point in the center of the arc and moving it rotates the arc's vector instead of moving the arc's center point. To access the center point, use *Shift+click* and the point selection arrows in the *Control Point* dialog, and use the *Object Move Point* to move the arc.

Like the parallelogram, the arc is simply a special case (sub-class) of the polygon, so it has the usual polygon attributes, like *LineType* and *FillColor*. In addition, an arc has the following attributes:

ArcFillType
Defines the type of the arc: GLG_CHORD, GLG_SECTOR or GLG_BAND.

AngleType
Defines the way the arc's angles are defined. Possible choices are GLG_START_AND_ANGLE and GLG_START_AND_END.

StartAngle and **EndAngle**
Define the angular position of the start and end points of the arc relative to its center. The angles are measured in degrees (counter clockwise). The StartAngle is always measured from the 3 o'clock position. The *EndAngle* is measured relative the *StartAngle* if *AngleType* is set to GLG_START_AND_ANGLE, and relative to the 3'o'clock position if the value of *AngleType* is GLG_START_AND_END. A circle is represented as an arc with a start angle of 0° and an end angle of 360°.

Radius
Defines a radius of an arc's curved edge.

MinRadius
Defines the inner radius of an arc band for arcs of the GLG_BAND type.

Resolution
The arc's resolution is the number of line segments used to render its perimeter. A circle drawn with a resolution of 5 is simply a regular pentagon. The default value for this is 100. A smaller value may be used for small arcs and circles to increase performance.

As with the parallelogram, an arc may be exploded in the Builder into its constituent polygon. The center and vector control points disappear, and simple polygon control points are shown on the arc's perimeter.

Spline

A **spline** is a multi-point Bezier or Catmull Rom cubic spline used to render curves in 2D or 3D space. A one-segment spline is a parametrically represented curve controlled by 4 control points. The shape of the segment may be changed by dragging the control points. The multi-point spline is a "blending" of one or more spline segments with a variable number of points. The spline starts and ends at the first and last control points respectively. The intermediate control points control the curvature and shape of the spline.

Like the parallelogram, the spline is a special case of the polygon, so it has the usual polygon attributes, like *LineType* and *FillColor*. The spline has two additional attributes:

SplineType
Defines the type of a spline, GLG_B_SPLINE (Bezier) or GLG_C-SPLINE (Catmull Rom). The Catmull Rom spline passes through the control points, while the Bezier spline yields a smoother curve due to its continuous second derivative.

SplineResolution
The spline's resolution is the number of line segments used to render each spline segment. The default value for this is 10. A larger value may be used to increase the rendering quality of splines with large segments or high curvature.

Text

The text object is used to place labels and legends in a GLG drawing. There are three types of text object, which differ both in their behavior and in the number of control points:

FIXED
Has one control point defining its position. The text font size is defined by the *FontSize* attribute, and is not controlled by a bounding rectangle.

SCALED
Text has two control points defining a rectangle in which to fit the text. The *FontSize* attribute defines the maximum and the *MinFontSize* attribute controls the minimum size allowed. The actual size of the font is determined dynamically by constraining the text to the rectangle. Note that the scaling done is not infinitely variable: different sizes of text are selected from the fonts in the font table according to the size of the bounding rectangle. Setting *MinFontSize* to -1 may be used for automatic text label decluttering, in which case the text will disappear if the drawing is zoomed out of and there is not enough space to draw the text inside the rectangle.

SPACED
Text has three control points. The letters of the text are evenly distributed along the line connecting the first two points. For multi-line text, the third point controls the position of the text's lines. Similar to the SCALED text, the SPACED text is fit to the parallelogram defined by it's three control points, with the *FontSize* and *MinFontSize* defining the maximum and minimum font sizes for scaling.

All text objects have the following attributes (in addition to control points):

TextColor
Defines the color of the text.

Compatibility Note: The name of this attribute changed in the release 2.9. Previously, the default attribute name for this attribute was *EdgeColor*. The *GlgCompatibilityMode* global configuration resource may be set to obtain behavior compatible with the earlier versions.

TextString
This is the text string displayed by the text object. If the text string contains the \n (ASCII NL:) and \r (ASCII CR) characters, they will be used as line separators and the text will be displayed in multiple lines. The *TextString* of a multi-line text can be edited only by using the text edit field of the *Attribute* dialog (ellipsis button ...).

TextType
Can have the following values:
FIXED (GLG_FIXED_TEXT)
SCALED (GLG_AUTOSCALED_TEXT)
SPACED (GLG_SPACED_TEXT).

Direction
Defines whether text is GLG_HORIZONTAL, GLG_VERTICAL, GLG_VERTICAL_ROTATED_RIGHT or GLG_VERTICAL_ROTATED_LEFT.

Anchor
Defines vertical and horizontal text alignments relative to the text's control point for the FIXED text, or within the text bounding box for other text types. The choices are CENTER, LEFT, or RIGHT for horizontal alignment, and CENTER, TOP, or BOTTOM for the vertical.

The corresponding defined constants are GLG_HCENTER, GLG_HLEFT, GLG_HRIGHT and GLG_VCENTER, GLG_VTOP, GLG_VBOTTOM. The two choices are combined (logical OR) to define the resource value.

FontType
Specifies the type of the font used to draw the text. *FontType* is a font family index in the viewport's font table. Refer to the *Editing a Font Table* section on page 147 for information on adding font types to the viewport's font table.

FontSize
Defines the font size of the FIXED text, or the maximum font size to use for fitting other text types. *FontSize* is a font size index to the viewport's font table. Refer to the *Editing a Font Table* section on page 147 for information on adding font sizes to the viewport's font table.

MinFontSize

Specifies the minimum font size to use when fitting the text.The *MinFontSize* is a column index to the viewport's font table. Setting *MinFontSize* to -1 activates automatic decluttering feature for SCALED and SPACED text types. In the case the text will not be drawn if there is not enough space to fit the smallest font (0) into the text area.

SizeConstraint

This attribute is used to synchronize the fitting of SCALED and SPACED text objects. If more than one text object is used, fitting may yield different font sizes due to different area sizes or different string lengths of each text object. As a result, they will be rendered using different font sizes.

The actual value of this attribute is irrelevant, because it is determined dynamically. However, if the attribute is constrained, all text objects with constrained *SizeConstraint* attributes will vary together, using the same font size regardless of the string length and other conditions. The *MinFontSize* for all constrained text objects has to be set to the same value as well.

If several text objects have constrained *SizeConstraint* attributes, they all will be displayed using the smallest needed font size in the group. To avoid constraint loops, the font size will stay small until the drawing is resized, even if the original reason for using the small font size has been eliminated.

Text Box

A slot for attaching an optional *Box Attributes* object to control attributes of an optional box drawn around a text object. A filled box may be used to provide a background for drawing a text object. The box attributes object is accessible by using the *Add/Edit Text Box* buttons in the *Object Properties* dialog. To delete box attributes, use the *Delete Text Box* button in the *Box Attributes'* properties. The attribute assumes the value of NULL if no box attributes object is attached, in which case no box is drawn. See *the BoxAttributes* section on page 143 for details.

When a box attributes object is added to all text objects in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added box attribute objects (the attributes are constrained if the default *Constrained Clone* setting is used).

Rendering

A slot for attaching an optional *Rendering* object to control an expanded set of rendering attributes, such as gradient fill, cast shadows, fill level and arrowheads. The rendering object is accessible by using the *Add/Edit Rendering* buttons in the *Object Properties* dialog. To delete the rendering object, use the *Delete Rendering* button in the Rendering Object properties. The attribute assumes the value of NULL if no rendering object is attached. See *the Rendering* section on page 140 for details.

When a rendering object is added to all objects in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added rendering objects (the attributes are constrained if the default *Constrained Clone* setting is used).

Note that a text object's behavior under various graphical transformations is limited by the availability of suitable fonts. For reasons of efficiency, GLG text objects contain text displayed in un-transformed fonts, taken from the viewport's font table. This limits a text object's ability to react appropriately to shear and scale transformations, for example, but greatly improves real-time update performance by eliminating multiple instances of similar fonts scaled slightly differently.

A viewport's font table can be modified to add custom fonts, as well as increase the number of available font sizes to expand text scaling limits. Refer to the *Editing a Font Table* section on page 147 for details.

Marker

The marker object is used to mark a position in space. For example, a point graph might be presented as a collection of marker objects arranged on a graph. It has a fixed size, and does not change when a window is resized.

Markers have only one control point defining their position. A marker object also has the following attributes:

MarkerType
Defines the shape of the marker. When drawn, a marker consists of components like cross, rectangle, filled rectangle, circle, filled circle, diamond and dot. Any mix of components can be chosen to represent a marker as defined by the marker type. The components are chosen from the following list:

CROSS
SQUARE
FILLED SQUARE
CIRCLE
FILLED CIRCLE
DOT
DIAMOND
FILLED_DIAMOND

The marker drawn is a superposition (Logical OR) of the set chosen by the *MarkerType* value.

MarkerSize
Defines the size of the marker in pixels.

FillColor and **EdgeColor**
Define colors used to draw marker's components.

AntiAliasing
Controls antialiasing of the marker's edges and is the same as the AntiAliasing attribute of a polygon. In the OpenGL environment, the GLG_ANTI_ALIASING_DBL setting may be used for smoother scrolling of plots with markers in a real-time chart.

Rendering
A slot for attaching an optional *Rendering* object to control an expanded set of rendering attributes, such as gradient fill, cast shadows, fill level and arrow heads. The rendering object is accessible by using the *Add/Edit Rendering* buttons in the Object Properties dialog. To delete the rendering object, use the *Delete Rendering* button in the Rendering

Object properties. The attribute assumes the value of NULL if no rendering object is attached. See *the Rendering* section on page 140 for details.

When a rendering object is added to all objects in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added rendering objects (the attributes are constrained if the default *Constrained Clone* setting is used).

Image

The image object is used to represent graphical images in GIF, JPEG, PNG and BMP formats. The GIF, JPEG and PNG formats are supported across all platforms, while the BMP format is supported only in the C/C++ on Windows, as well as in the Java and C#/.NET versions. TIFF images are also supported in the Java and C#/.NET versions of the Toolkit. Transparent colors are supported via the *TransparentColor* attribute, and image transparency is supported via the *Visibility* attribute (by setting it to fractional values). In the OpenGL, Java and C#/.NET environments, PNG images with an alpha channel transparency are also supported.

An image may be of fixed size, defined by the size of the image in the original file, and have one control point. The image object may also be resizable, in which case it's size is adjusted to fit the rectangle defined by the image's two control points.

An image object has the following attributes:

ImageType
Defines the type of the image: a fixed-size (GLG_FIXED_IMAGE) with 1 control point or a scalable (GLG_SCALED_IMAGE) with two control points.

ImageFile
Defines the location of the image file in one of the supported image formats. In the Graphics Builder, as well as for the C/C++/ActiveX, the type of the file is determined by the file's extension:

.gif for a GIF file
.jpg for a JPEG file
.png for a PNG file
.bmp for a Windows bitmap (Windows only).

Note: When an image object is created in the Graphics Builder, a file browser is used to select an image file, and an absolute path is stored in *ImageFile*. To allow the application to be moved to a different directory or a different environment (web or Java) without adjusting image paths, it is recommended to edit the stored *ImageFile* path to make it relative to the location of the drawing.

If the *ImageFile* attribute defines a relative file name, the Toolkit tries to find the file it in the following order:

- attempting to load the file relative to the directory of the drawing
- trying to locate the file in one of the directories defined by the GLG_PATH environment variable or the *GlgSearchPath* global configuration resource
- attempting to load the file relative to the current directory as the last resort.

A *List* transformation may be attached to the *ImageFile* attribute to specify a list of image files for implementing image dynamics.

Cross-Platform Use Note: For cross-platform, Java and web-based deployment, use '/' as a path delimiter even on Windows. On Windows, the Builder converts '/' to '\' automatically when necessary.

Anchor
Defines the vertical and horizontal alignments of the fixed size image relative to its control point. The choices are CENTER, LEFT, or RIGHT for horizontal alignment, and CENTER, TOP, or BOTTOM for the vertical.

The corresponding defined constants are GLG_HCENTER, GLG_HLEFT, GLG_HRIGHT and GLG_VCENTER, GLG_VTOP, GLG_VBOTTOM. The two choices are combined (logical OR) to define the resource value. The attribute has no effect for scalable images.

TransparentColor
Defines the image color to be rendered as transparent. This may be used for rendering icons with transparent background. The color RGB values are specified using the default 0-1 range, or using 0-255 range if the *255 Color Display* option is activated. All image pixels with this RGB color value will be rendered as transparent. By default, the attribute is set to a value which disables transparency: (-1,-1,-1) in the default color mode, or (-255,-255,-255) value in the *255 Color Display* mode.

When a transparent GIF image is deployed in a GLG drawing in Java and C# environments, both the transparent color defined by the *TransparentColor* property and the transparent color defined in the GIF image are enabled.

In the OpenGL, Java and C#/.NET environments, PNG images with an alpha-channel transparency may also be used with or without the use of the *TransparentColor* attribute.

Windows GDI Note: For transparent GIF images with the GDI driver on Windows, the *TransparentColor* setting overrides the transparent color defined in the GIF image. If *TransparentColor* is set to the disabled value described above, the transparent color defined in the GIF image is used.

Windows Note: the transparent color mode is supported on OS versions that support the *TransparentBlt* method.

GIS Object

Generic Logic provides a GIS Map Server product designed for real-time rendering of high-resolution maps with millions of objects. There are a variety of applications that might need to display a map in the background as contextual information, and place GLG objects on top of the map to represent dynamic or static icons. Such applications need to provide map zooming and panning functionality, handle window resizing and user interaction with the icons.

The GIS Object seamlessly integrates Map Server functionality into the GLG drawing, both in the application and in the Graphics Builder. The GIS Object displays a map image in a selected GIS projection and transparently handles all aspects of interaction with the Map Server, automatically issuing map requests every time the map is resized, panned or zoomed.

The GIS Object supports **integrated zooming and panning**, as well as integrated scrollbars. In the *GIS Zoom Mode*, zoom and pan controls zoom and pan the map displayed in the GIS Object instead of zooming and panning the viewport's drawing. The map can also be **dragged with the mouse**, which works best with either fast CPUs or not very complex maps. In the Builder, the *GIS Zoom Mode* may be set by using the *Arrange, GIS Zoom Mode, Set as parent viewport's GIS Object* menu option while the GIS Object is selected. The *GIS Zoom Mode* is persistent and is saved with the drawing.

The map displayed in the GIS Object can also be zoomed and panned programmatically via the *GlgSetZoom* method. Refer to the description of the *Pan* and *ZoomEnabled* attributes of a viewport object on page 85 and page 86 correspondingly for details of the integrated GIS Zooming.

The GIS Object can be used as a container that holds dynamic icons, polylines and other graphical objects. The objects added to the GIS Object are drawn on the map in the **GIS Rendering Mode**, in which the X and Y coordinates of the objects' control points are interpreted as degrees of longitude and latitude, and Z coordinate is interpreted as an elevation above the Earth surface in meters. This allows positioning of icons and lines on the map by defining their lat/lon coordinates directly, without any coordinate conversions. When the map is zoomed or panned, the objects drawn on the map will be automatically adjusted to zoom and scroll with the map. The GIS Object also provides utilities to convert from screen or world coordinates to latitude/longitude in the selected projection and vise versa.

The Graphics Builder supports the **GIS Editing Mode** for interactive creating and editing objects drawn on the map. In this mode, dynamic icons, polylines and other objects can be drawn or positioned on the map with the mouse in the lat/lon coordinates. The Builder automatically converts the mouse position from the screen to lat/lon coordinates, which are stored in the object's control points. The Builder transparently handles GIS projections, which allows the user to draw polylines on top of the globe displayed in the orthographic projection. To start the *GIS Editing Mode*, select the GIS Object, then press the *Hierarchy Down*  button to go down into it. In the *GIS Editing Mode*, you can draw and position objects on top of the map with the mouse, as well as edit attributes of the previously created objects. All objects added to the GIS Object in the *GIS Editing Mode* will be contained in its *GISArray* and will be saved with the GIS Object. Dynamic icons and other graphical objects may also be added to the GIS Object programmatically at run time using one of the *GlgAddObject* methods. The GLG GIS Demo and GLG AirTraffic Control Demo may be used as source code examples of adding dynamic icons at run-time.

The two control points of the GIS Object define the size of the map image. When the GIS Object is used to provide a background map for the drawing, its points' values may be set to (-1000, -1000, 0) and (1000, 1000, 0) to cover the whole drawing (in the Builder, *Shift-click* on the control point with the mouse to enter exact coordinates).

The following attributes of the GIS Object provide easy resource-based access to the underlying Map Server functionality (refer to the *GLG Map Server Reference Manual* for more information):

FillColor
: Defines the color of the map's background, which is visible when the map is sufficiently zoomed out.

GISDisabled

Disables the GIS Object for quick editing.

GISProjection

Defines the projection used to render the map: GLG_RECTANGULAR_PROJECTION or GLG_ORTHOGRAPHIC_PROJECTION. The rectangular projection displays the world as a rectangular region and is convenient for displaying detailed maps, where parallels and meridians appear as straight lines. The orthographic projection maps the whole world onto a sphere, and is often used for the top-level globe-like views.

GISCenter

Defines the latitude and longitude of the map to be displayed in the center of the GIS Object. This attribute is of the geometrical (G) type and is a set of three values. The first two values supply the longitude and latitude correspondingly, while the third value must be set to zero. The attribute is automatically adjusted when integrated GIS panning is performed.

If the GIS object with a map in the RECTANGULAR projection is clipped by the viewport's visible area, the actual used center may be different from the value of the *GISCenter* attribute. The actual used center may be queried at run time using the *GISUsedCenter* resource (G).

GISExtent

Defines the extent of the map visible in the GIS Object. This attribute is of the geometrical (G) type and is a set of three values. The first two values supply the X and Y extents, while the third value must be set to zero. For the **rectangular** projection the extents are measured in degrees of the longitude and latitude. For the **orthographic** projection the extents are specified in meters as dictated by the Open GIS Standard. The default extent value for the orthographic projection is 14,000,000, which is slightly bigger than the earth's diameter. The attribute is automatically adjusted when integrated GIS zooming is performed.

If the GIS object with a map in the RECTANGULAR projection is clipped by the viewport's visible area, or if the GIS object's *GISStretch* is set to OFF, the actual used extent may be different from the value of the *GISExtent* attribute. The actual used extent may be queried at run time using the *GISUsedExtent* resource (G).

If a GIS object with a map in the ORTHOGRAPHIC projection is clipped by the viewport's visible area, its actual GISExtent is not adjusted to include only the visible area of the map. Instead, the original GISExtent defined in the GIS object is used to define the projection parameters without any adjustments, while the map image is generated only for the visible part of the map for efficiency. This makes it possible to generate zoomed images of the globe with a visible horizon line, as shown in the Trajectory demo. The demo uses a GIS object bigger than the viewport visible area to achieve the desired visual appearance of the map.

GISAngle

Defines the map rotation angle in degrees. For example, an application can set the rotation angle to display a map from the point of view of an airplane pilot. The map rotation feature is supported only in the rectangular projection; the attribute's value is ignored in the orthographic projection. The attribute is automatically adjusted when integrated GIS rotation is performed.

GISStretch

Defines the stretch mode. If the attribute is set to YES, the map is stretched, otherwise the aspect ratio of the map is preserved and the map image may include an area which is slightly bigger than the area defined by the *GISExtent* attribute.

GISDataFile

Specifies a Server Data File (.sdf) which describes the dataset to be used by the Map Server to generate the map image. It may be specified using either a relative or absolute path. This attribute is used only by the C/C++ and ActiveX version of the Toolkit, as well as the Graphics Builder, which use the Map Server in the form of a C library.

Cross-Platform Use Note: Use a relative path instead of an absolute path to allow the application to be moved to a different directory or a different environment (web or Java) without adjusting the paths. For cross-platform deployment, use '/' as a path delimiter even on Windows. On Windows, the Builder converts '/' to '\' automatically when necessary.

GISMapServerURL

Specifies the Map Server URL to be used by the Java and C#/.NET versions of the Toolkit. The URL has to have a web server based GLG Map Server setup as described in the *GLG Map Server Reference Manual.* This attribute only affects the Java and C#/.NET versions of the Toolkit, which connects to a web-based GLG Map Server to retrieve the map image. The GIS Object handles all aspects of connecting to the Map Server URL with proper parameters.

GISLayers

Defines a list of layers to be displayed in the generated image. The value of this attribute is a comma separated list of layer names, defined in the Server Data File (.sdf). The "default" string may be used to enable the layers whose "DEFAULT ON" attribute is set to YES in the Layer's Information File (.lif). See the *GLG Map Server Reference Manual* for details.

GISArray

A group object used as a container to hold graphical objects that will be drawn on top of the map. GLG objects added to the GIS Object in the Builder are placed into this group. The content of the group is rendered in the *GIS Rendering Mode*, which interprets coordinates of the objects' control points as lat/lon coordinates. The objects may be added programmatically either to the GIS Object or directly to its *GISArray.* When objects are added to the GIS Object, they are placed into its *GISArray* group.

GISVerbosity

May be set to a value from 0 (no debugging output) to 10 (maximum debugging output) to assist debugging of the Map Server setup. It may also be set to a negative value in the range from -1 (overall performance data) to -3 (the most detailed per-tile performance data) to display performance measuring information. The attribute may also be set to a value of 1001 or 1002 to display tile extents. This attribute has no effect in the Java and C#/.NET versions of the Toolkit.

GISDiscardData

Controls Map Server data caching. If set to NO (default), the Map Server data is cached resulting in faster image generation. For map images that are generated only once or very infrequently, the attribute may be set to YES to discard data after generating the image, saving memory. This attribute has no effect in the Java and C#/.NET versions of the Toolkit.

The GIS Object may be prototyped in the Builder by going down into it using the *Hierarchy Down*  button. The zoom and pan controls may be used to zoom and pan the map, testing the automatic layer switching of the GLG Map Server and the map server setup. Alternatively, the *GIS Zoom Mode* may be set by using the *Arrange, GIS Zoom Mode, Set as parent viewport's GIS Object* menu option. With the *GIS Zoom Mode* activated, the map in a viewport can be zoomed and scrolled with the Builder's zoom and pan controls without going down into the GIS Object. The viewport's *Pan* property may be set to *Pan XY* to use the viewport's integrated scrollbars for scrolling the map.

The icons and other GLG objects drawn on the map in the *GIS Rendering Mode* are clipped to the visible area of the map, which eliminates icons on the invisible part of the globe in the ORTHOGRAPHIC projection. In the ORTHOGRAPHIC projection, the polyline segments on the invisible part of the globe are also eliminated. When rendering polylines that span the whole globe in the ORTHOGRAPHIC projection, it is recommended to use a sufficient number of points for better rendering of polyline segments that cross the boundary between the visible and invisible parts of the globe.

Refer to the *GLG Map Server Reference Manual* for more information on the GLG Map Server, its setup and usage.

Viewport

A viewport object is a GLG encapsulation of a window, and is rendered as a non-transparent rectangular region into which graphical objects can be placed. You can think of it as the drawing surface for a GLG drawing. Unlike a simple rectangle, however, the viewport object contains the objects that appear in front of it (which can include other viewports). This creates a convenient way to group objects in a GLG drawing. A viewport can control the resizing of its member objects, as well as the magnification, angle, and lighting with which a drawing is seen.

A viewport also differs from a simple rectangle in that it always appears in the plane parallel to the screen. You cannot view a viewport from an oblique angle.

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.

To add objects to the viewport, the editing focus has to be moved in the viewport, by either going "down" into the viewport using the *Hierarchy Down* button  , or by setting the editing focus by *Ctrl-Shift*-clicking on the viewport. When finished, use the use the *Hierarchy Up* button  or the *Main Focus* button  , depending on the action used to set the focus inside the viewport.

Note: If the focus was moved into the viewport, the *Hierarchy Down* button will be disabled. To traverse down into the viewport's objects, use the *Hierarchy Down* button to get inside the viewport, and then use it again to get inside the viewport's objects.

A **widget** is defined to be a viewport that has resources (*HasResources* is YES) and is named *$Widget*. When the GLG API reads a drawing from a file, it looks for a widget definition to use. The widget name should appear only once in a drawing. All subsequent resource read and set operations implicitly refer to this object. In an application, a viewport could be used to define a graph or a control.

A viewport's attributes may be divided into four categories. The first group of attributes controls the appearance of the viewport's background rectangle, and the second controls the display of its child objects. A third group of attributes controls some aspects of event handling and viewport interactive behavior. The last group is made up of window-specific attributes, and is embodied by the screen object, described in the next section.

In addition to the two control points, the viewport backing rectangle has the following attributes:

FillColor
Defines a background color of the viewport.

EdgeColor
Defines a border color for the viewport rectangle.

LineWidth
Defines the viewport rectangle's border width.

ShadowWidth
Defines the width of shadows. If the value differs from 0, the shadow bevels are drawn around the borders of the viewport. The sign of the *ShadowWidth* controls the type of the bevels: raised shadows for positive values and depressed shadows for negative values. This attribute is inherited from the viewport's screen object described below.

Pan
The *Pan* attribute controls integrated scrolling. If panning is activated, the viewport displays scrollbars and handles scrolling when the drawing extends beyond the viewport's visible area. In the GIS Zoom Mode, the scrollbars control scrolling of the map displayed in the viewport's GIS Object, and in the Chart Zoom Mode, they control chart scrolling.

Possible values are:

NONE
Disables pan scrollbars.

PAN X

PAN Y

PAN XY
Enables either X or Y scrollbar, or both X and Y scrollbars.

AUTO PAN X

AUTO PAN Y

AUTO PAN XY
Automatically enables either X or Y scrollbar, or both X and Y scrollbars. The scrollbars will automatically appear only when the content of the viewport (or content of the chart in the Chart Zoom Mode) extends outside of the visible area and may need to be scrolled.

PAN X & AUTO PAN Y

PAN Y & AUTO PAN X
Enables a permanent scrollbar in one direction and an automatic scrollbar in another direction. The automatic scrollbar will appear as needed when the content extends outside of the visible area and may need to be scrolled.

When the scrollbars are enabled, they may be accessed as the *GlgPanX* and *GlgPanY* resources of the viewport. When both scrollbars are enabled, a viewport object named *GlgPanSpacer* is also created to cover the lower right corner area between the scrollbars.

ActivePan

Read-only attribute, contains a bit mask composed of the GLG_PAN_X and GLG_PAN_Y binary flags indicating which scrollbars are currently displayed.

ZoomEnabled

The *ZoomEnabled* attribute enables keyboard accelerators for integrated zooming and panning. When it is set to YES, pressing an accelerator key performs a corresponding zooming or scrolling operation. This setting is primarily used for quick interactive testing and prototyping in the Graphics Builder.

If the attribute is set to NO, keyboard accelerators are disabled, but zooming and panning operations can still be performed via the *GlgSetZoom* API function. This is the preferred method for a run-time application, where zooming and panning operations are performed via the interface buttons, while the keyboard accelerators are disabled to prevent accidental use.

The following accelerator keys are supported:

u - pan up

d - pan down

l - pan left

r - pan right

i - zoom in (zoom in in the X/Time direction in the Chart Zoom Mode)

I - zoom in in Y direction (Chart Zoom Mode only)

o - zoom out (zoom out in the X/Time direction in the Chart Zoom Mode)

O - zoom out in Y direction (Chart Zoom Mode only)

n - reset zoom.

In the GIS Zoom Mode with map in the ORTHOGRAPHIC projection, resets zooming, but keeps the GIS center unchanged. In the RECTANGULAR projection, resets zooming and paning, but keeps the rotation angle unchanged.

In the Chart Zoom Mode, resets the Y ranges to fit all chart plots in the visible area of the chart in Y direction.

N - reset zoom.

In the Chart Zoom Mode, resets the Y ranges and also changes the chart's X span to show all accumulated data samples in the visible area of the chart.

Shift-click-drag - ZoomTo, same as *'t'*, see details below.

t - start generic ZoomTo mode (left-click and drag the mouse to finish)

In the Chart Zoom Mode, if the first point of the ZoomTo box is located within the X or Y axis area, zooming will be performed only in the direction of the selected axis. For example, if the user defines the ZoomTo box in the X axis area, the chart will be zoomed only in the X direction.

_ - start ZoomToX mode, which zooms only in the X direction and preserves the Y scale (left-click and drag the mouse to finish). It is especially useful in the Chart Zoom Mode.

| - start ZoomToY mode, which zooms only in the Y direction preserves the X scale (left-click and drag the mouse to finish). It is especially useful in the Chart Zoom Mode.

@ - start ZoomToXY mode (left-click and drag the mouse to finish)

T - start custom ZoomTo mode. A custom zoom mode lets the user define the ZoomTo area without performing the zoom operation. An application can use the selected ZoomTo rectangle as the input to implement custom zooming or object selection logic.

e - abort ZoomTo mode

Control-click-drag - Drag the drawing or map with the mouse, same as *'s'*, see details below.

s - start generic dragging mode (left-click and drag the drawing with the mouse to finish).
In the Chart Zoom Mode, if the user clicks and drags the mouse within the X or Y axis area, scrolling will be performed in the direction matching the direction of the selected axis.

^ - start vertical dragging mode (left-click and drag the drawing with the mouse to finish). It is especially useful in the Chart Zoom Mode.

> - start horizontal dragging mode (left-click and drag the drawing with the mouse to finish). It is especially useful in the Chart Zoom Mode.

& - start XY dragging mode (left-click and drag the drawing with the mouse to finish).

f - fit the drawing to the visible area of the viewport (Drawing Zoom Mode only)

F - fit the area of the drawing defined by an object named *GlgFitArea* to the visible area of the viewport (Drawing Zoom Mode only)

U - anchor on the upper edge of the drawing (Drawing Zoom Mode only)

D - anchor on the lower edge of the drawing (Drawing Zoom Mode only)

R - anchor on the right edge of the drawing (Drawing Zoom Mode only)

L - anchor on the lower edge of the drawing (Drawing Zoom Mode only)

A - rotate the drawing clockwise around X axis (Drawing Zoom Mode only)

a - rotate the drawing counterclockwise around X axis (Drawing Zoom Mode only)

B - rotate the drawing clockwise around Y axis (Drawing Zoom Mode only)

b - rotate the drawing counterclockwise around Y axis (Drawing Zoom Mode only)

C - rotate the drawing clockwise around Z axis
(Drawing Zoom Mode or GIS Zoom Mode with the rectangular projection only)

c - rotate the drawing counterclockwise around Z axis
(Drawing Zoom Mode or GIS Zoom Mode with the rectangular projection only)

g - If the mouse is located on top of a GIS Object, sets the viewport's GIS Zoom Mode and remembers the selected GIS Object. If the mouse is located on top of the chart object, sets the viewport's Chart Zoom Mode. In the GIS Zoom Mode, the map displayed in the GIS Object is zoomed and panned instead of the viewport's drawing, and in the Chart Zoom Mode, the chart is zoomed and scrolled. Zooming, panning, ZoomTo and reset accelerators are supported in the GIS and Chart Zoom Modes.

G - Resets the GIS or Chart Zoom Mode.

p - available only in the Graphics Builder in the GIS or Chart Zoom Mode.
In the GIS Zoom Mode, it displays the lat/lon and X/Y coordinates of the point at the current cursor position. If the GLG_GIS_ELEVATION_LAYER environment

variable is set to a valid elevation layer name, the point's elevation is also displayed.
In the Chart Zoom Mode, displays the X/Time value corresponding to the current cursor position, and the Y value in the range of the first Y axis.

q - available only in the Graphics Builder in the GIS or Chart Zoom Mode.
In the GIS Zoom Mode, displays information about the GIS selection. If GISVerbosity is set to 2000 or 2001, extended information is also written into the GLG error log file and printed to the terminal on Linux/Unix.
In the Chart Zoom Mode, it displays information about a data sample pointed by the cursor. The data sample is selected using the X mode of the chart's *TooltipMode* attribute.

Q - available only in the Graphics Builder in the Chart Zoom Mode.
It is the same as the 'q' accelerator, but uses the XY selection mode.

Setting *ZoomEnabled* to YES enables accelerators in the Run mode of the Builder and at run time. If *ZoomEnabled* is set to NO, the accelerators are disabled, but the integrated zooming and panning may still be used at run time programmatically by invoking the *GlgSetZoom* method. The *GlgSetZoom* method takes accelerator keys listed above as its zoom type parameter.

There are several accelerators for the ZoomTo operation, allowing to activate zooming in only X, Y, or in both X and Y directions. To perform the ZoomTo operation, press one of the zoom accelerators (*'t'*, *'_'*, etc.), then left-click and drag the mouse to define the area to zoom to. The user can also zoom to an area by holding the *Shift* key and then using the left mouse button to click and drag the mouse to define a zooming rectangle. Zooming in only X or Y direction is especially useful for real-time charts, allowing to zoom only along the time axis or the Y axis.

The are also accelerators for panning and scrolling the drawing by dragging it with the mouse. Several accelerators are provided, for panning and scrolling in only X, Y, or in both X and Y directions. In the GIS Zoom Mode, the map is scrolled by dragging it with the mouse, and in the Chart Zoom Mode, the chart is scrolled. To start, press one of the panning accelerators (*'s'*, *'>'*, etc.), then left-click and drag the mouse to scroll the content of the drawing. The user can also use the *Control-click-drag* sequence. Panning accelerators are primarily used for starting the dragging operation via the programming API at run time.

Performing zoom and pan actions on a viewport generates Zoom and Pan messages. Refer to the *Appendix B: Message Object Resources* section of the *GLG Programming Reference Manual* for details.

ProcessMouse
Controls the viewport's processing of the mouse events. The value of the attribute is formed by ORing binary masks to enable individual types of mouse events. Possible values may contain a combination of the following:

None (GLG_NO_MOUSE_EVENTS)
Disables processing mouse events for the viewport. May be used to reduce CPU consumption for viewports that do not need to process any events.

Tooltip (GLG_MOUSE_OVER_TOOLTIP)
Enables object tooltips. Use *Object, Add Tooltip* menu option to add a tooltip to an object. Custom tooltip formatters can be supplied via the *GlgSetTooltipFormatter* method to generate dynamic context-based tooltip strings on the fly. Button tooltips are always active regardless of the setting of the *ProcessMouse* attribute.

Tooltip (Named Objs) (GLG_MOUSE_OVER_TOOLTIP | GLG_NAMED_TOOLTIP)
Enables tooltips for all named objects. The object's name will be used as a tooltip string.

Click (GLG_MOUSE_CLICK)
Enables processing of the mouse click events in the viewport. It activates processing of actions with *Trigger=MOUSE_CLICK*, as well as object selection messages on the mouse click, which are passed to the Input callback at run time. It also activates the old-style (prior to v. 3.5) custom object selection events and mouse click feedback controlled by the *MouseClickEvent*, *MouseClickState* and *MouseClickToggle* properties of an object.

Move (GLG_MOUSE_OVER_SELECTION)
Enables processing of the mouse over events in the viewport. It activates processing of actions with *Trigger=MOUSE_OVER*, as well as object selection messages on the mouse over, which are passed to the Input callback at run time. It also activates the old-style (prior to v. 3.5) custom mouse over events and mouse over feedback controlled by the *MouseOverEvent* and *MouseOverState* properties of an object.

Masks of a viewport's *ProcessMouse* attribute are inherited by its children viewports. For example, setting *ProcessMouse* to *Click* will enable mouse click processing for the viewport, as well as for all of its children viewports. If a viewport's *ProcessMouse = Click,* and the *ProcessMouse* of its child viewport is set to *Move*, the child viewport will process both the mouse click and mouse over events.

Refer to the *Integrated Features of the GLG Drawing* chapter on page 45 for details on custom events, mouse feedback and object tooltips.

Refer to the description of the GlgDisablePre35ObjectEvents global configuration variable on page 179 for information on disabling the old-style custom events and tooltips (prior to v. 3.5), which may decrease CPU load when moving the mouse over a large drawings.

Handler
A viewport may become an input widget (or control) by naming an input handler with this attribute. The *Handler* attribute identifies the style of control that is adopted, such as slider, knob, switch, and so on. The use of input handlers is described in *Input Objects*.

DisableInput
Controls whether or not a viewport and its input handler react to input events. Setting the attribute to YES disables all input events in the viewport; if the viewport has an input handler attached, it also disables the handler. The YES setting also disables all children viewports, except for the Java/Swing environment.

DepthSort
The *DepthSort* attribute defines how to render overlapping objects inside the viewport by controlling hidden surface removal. A hidden surface is one whose view is fully or partially blocked by another object. For example, a drawing might consist of two circles. If you look at the circles from an angle where the position of one circle is between the viewer and the other circle, the blocked circle will not appear to be drawn if the objects are depth-sorted.

If a viewport with *DepthSort*=NO contains several groups with different settings of the *DepthSort* attribute, the groups themselves will be drawn in the natural order, but the objects inside groups with *DepthSort*=YES will be rendered using the hidden-surface removal.

If the OpenGL driver is used, setting the attribute to YES or SPECIAL activates the OpenGL depth buffer to perform hidden surface removal for objects in the viewport. Setting the attribute to NO or any other value disables OpenGL hidden surface removal.

When hardware acceleration is provided by a graphics card, the OpenGL-based hidden-surface removal yields real-time 3D performance, making it possible to render complex 3D drawings in real time. The hidden-surface removal works on the pixel basis and properly renders intersecting objects.

The OpenGL driver can be used for the GLG Graphics Builder, the GLG HMI Configurator, as well as for GLG applications using the GLG C/C++ library. On Windows, the OpenGL driver can also be used for GLG applications that use the GLG ActiveX Control.

If nested groups (or other objects) with *DepthSort*=NO are encountered while a parent's OpenGL-based hidden surface removal is active, these objects will be drawn in a separate pass after all objects with *DepthSort*=YES have been rendered and will appear on top.

If OpenGL hidden surface removal is active, any semi-transparent objects must be rendered last, on top of all opaque objects, to achieve expected transparency effect, which is an established OpenGL technique. In GLG, this can be easily achieved by placing all semi-transparent objects in a group with *DepthSort*=NO, which will cause the group to be drawn last, on top of all other objects.

The *GlgOpenGLZSort* global configuration resource controls the number of passes used by the OpenGL hidden surface removal. The default setting of 2 enables two-pass technique which helps to eliminate pixel artifacts for polygons that have both fill and edges. The edges are rendered in a second pass using an offset defined by the *GlgOpenGLDepthOffset* global configuration resource (100 be default). If polygons have only fill or only edges, *GlgOpenGLZSort* can be set to 1 to use a single pass for increased performance. Setting the resource to 0 disables OpenGL hidden surface removal and resorts to the slower non-OpenGL depth-sorting technique. Refer to page 345 in *Appendices* for more information.

When the OpenGL hidden surface removal is active, the fill of polygons is not anti-aliased. To anti-alias polygon edges, use polygons with *FillType*=FILL_EDGE, and *Shading*=FILL_EDGE_SHADING. The two-pass technique described above helps eliminate polygon edge artifacts.

If the GDI driver is used, setting the attribute to YES or SPECIAL activates a depth-sorting algorithm that renders objects in the order which depends on their position in the 3D space. Setting the attribute to NO or any other value disables depth sorting.

The SPECIAL setting uses a faster depth sorting algorithm which uses objects' bounding boxes for determining the drawing order of objects. The YES setting performs slower and more detailed tests. The algorithm used to sort objects is known not to work in complicated cases when objects intersect. Since any depth-sorting algorithm slows down the update procedure for a drawing, use it only when necessary.

The INHERIT and PARTS settings of the attribute have no meaning for viewports, but are used for other objects that use this attribute, such as groups. A viewport cannot inherit the *DepthSort* attribute.

KeepEditRatio

If set to YES, preserves the X/Y ratio of the viewport while editing its content in the Builder by going down into it using the *Hierarchy Down* button.

For example, the width of a viewport representing a toolbar is much bigger than its height. When *Hierarchy Down* button is used to go down into the viewport to edit its content, the viewport is extended to the entire drawing area, which stretches its content due to the different X/Y ratio of the drawing area. If *KeepEditRatio*=YES, the X/Y ratio of the viewport is maintained for the duration of editing by temporarily changing the SpanX and SpanY attributes of the viewport's screen. The span attributes are restored when going back up. The extent of the viewport's span is annotated by the round red markers in the corners of the default span area.

OwnsInputCB

Controls how the input callback is invoked for the input events occurring in the viewport. In an application code, an input callback is often attached to a top-level viewport. When an event occurs in a child viewport, the input callback is invoked with the *viewport* parameter set to the top-level viewport the callback is attached to.

However, the application may need to receive information about the actual viewport where the event occurred. In this case, the *OwnsInputCB* parameter of the child viewport may be set to YES, causing the input callback to be invoked with the *viewport* parameter set to the child viewport instead of the top-level viewport the input callback is attached to.

This makes it easier to handle commands in the application code, for example commands that display popup dialogs. These commands contain resource path to the dialog to be shown, and this path may be relative to the currently displayed page. To use a relative path when processing the command in the input callback, the application needs to know the viewport object of the current page. If each page has *OwnsInputCB* flag set to YES, the *viewport* parameter of the input callback will be set to the page's viewport object when the command is triggered. This avoids a need to attach an input callback to each page's viewport, as demonstrated in SCADA Viewer demo.

When the value of the *OwnsInputCB* is set programmatically, it follows the rules for attaching the input callback and must be set before hierarchy setup.

Light

A slot for attaching an optional *Light* object to control the viewport's lighting and 3D shading. The *Add/Edit Light* button may be used to add a light object to the viewport or edit its attributes if it already exists. To delete the Light object, use the *Delete Light* button in the Light Object properties. See the *Light Object* section on page 149 for details.

When a Light object is added to all viewports in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added Light objects (the attributes are constrained if the default *Constrained Clone* setting is used).

Rendering

A slot for attaching an optional *Rendering* object to control an expanded set of rendering attributes for the gradient fill. Other rendering attributes, such as cast shadows, fill level and arrow heads, are ignored for the viewport. The rendering object is accessible by using the

Add/Edit Rendering buttons in the Object Properties dialog. To delete the rendering object, use the *Delete Rendering* button in the Rendering Object Properties. The attribute assumes the value of NULL if no rendering object is attached. See *the Rendering* section on page 140 for details.

When a rendering object is added to all objects in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added rendering objects (the attributes are constrained if the default *Constrained Clone* setting is used).

Zooming and Viewing Transformations

Each viewport automatically creates a *Matrix* transformation, which is used for zooming when the viewport is edited in the Builder as well as for integrated zooming and panning at run-time.

User-defined viewing transformations (e.g. *Scale*, *Move, Rotate, Shear)* may also be added directly to the viewport to implement user-controlled zooming, panning or 3D rotating functionalities, which otherwise would require creating a group to contain all objects in the viewport and attaching the transformations to the group. Attaching the viewing transformations directly to the viewport makes it more convenient by eliminating an extra group.

To add a viewing transformation, move the focus inside the viewport, make sure no objects are selected, display the viewport's properties and press the *Add Dynamics* button to add a viewing transformation. Notice that selecting the viewport and adding dynamics adds a transformation to the viewport, transforming its control points, while adding dynamics with the focus inside the viewport adds a viewing transformation which affects the way the objects in the viewport are drawn.

After adding a viewing transformation, the default zooming transformation of the viewport can also be accessed in the Builder. This matrix transformation must always be present, and the Builder prevents it from being deleted or reordered.You can give this transformation a name to access it from a program as a named resource.

ZoomFactor

A read-only attribute of the viewport showing the current zoom factor; may be used to implement custom decluttering (turning layers on or off depending on the zoom factor) or icons of constant size. This attribute is not displayed in the viewport's Property dialog, but is accessible as a resource.

XYRatio

A read-only attribute of the viewport showing the current X/Y ratio of the viewport's window. It may be used to implement icons of constant X/Y ratio for a viewport with *Stretch* enabled, so that the icons will keep their X/Y ratio constant while other objects in the drawing will stretch. This attribute is not displayed in the viewport's Property dialog, but is accessible as a resource.

A viewport has a secondary screen object which is always attached to the viewport and controls its window-specific display attributes. Use the *Screen Attributes* button to display screen's properties. Screen attributes are inherited by the screen's viewport and may be accessed as resources of the viewport itself in an application.

Screen

The screen object is a mandatory child object of a viewport. It is designed to contain window specific attributes of a viewport, and is created automatically every time a viewport is created. It has the following attributes:

Double Buffering

Controls the usage of the double buffering for a screen. This may be set to NO or YES. When set to YES, screen updates are done incrementally off-screen, and all changed portions of the entire screen are updated at once. This usually creates a smoother illusion of motion than updating objects directly on the screen. Double-buffering is turned on by default. It can be useful to turn it off to see updates as they go in complicated drawings, such as a surface graph. If you have a lot of viewports, double buffering may turn to be an expensive resource, as it causes the window manager to allocate memory for the off-screen pixmaps. Turn it off if you want to decrease the amount of memory consumed.

Warning: Double buffering is known to cause problems when zooming with high zoom factors in a viewport that has other viewports in it. Zooming in the parent viewport increases the size of the child viewport. If the child viewport has double buffering set to YES, the excessive increase of the size causes the graphics server to exhaust memory trying to allocate a huge off-screen pixmap. The GLG Toolkit driver tries to catch this condition and disables double buffering, producing an error message if the viewport size becomes too big. But it may be too late in some cases. Turn double buffering off for the children viewports if planning to zoom into the parent viewport with big zoom factors.

When the OpenGL driver is used, the OpenGL's double buffering capabilities are used instead of the off-screen pixmap, which reduces memory consumption and increases rendering speed.

Resizable

Controls whether or not objects in the viewport are resized when the size of the viewport's window is changed. This attribute may be accessed programmatically using the *CoordSystem* resource name and may have the following values:

YES (WORLD) (`GLG_WORLD_COORD_SYSTEM` API constant)

A default GLG coordinate system used to define the objects' geometry. The extent of the visible part of the viewport is [-1000;+1000] in the world coordinates, and all objects in the viewport are resized when the viewport's size is changed. The world coordinate system's origin is positioned at the center of the viewport, with the X axis pointing to the right, the Y axis pointing up and the Z axis pointing to the viewer. The exact coordinate mapping is affected by the screen's *Stretch* and *PushIn* attributes as described below. The *SpanX* and *SpanY* attributes of the screen object may be used to change the extent of the visible portion of the viewport for advanced use.

NO (SCREEN) (`GLG_SCREEN_COORD_SYSTEM` API constant)

The screen coordinate system is used to define objects. In this coordinate system, coordinates are defined in screen pixels and objects' dimensions are not changed when the viewport is resized. The origin of the screen coordinate system is located at the top left corner of the viewport, with the X axis pointing right, the Y axis pointing down and the Z axis pointing away from the viewer to form a right-hand coordinate system for 3D rendering.

NO (GLG SCREEN) (`GLG_FLIPPED_SCREEN_COORD_SYSTEM` API constant)
Same as the **NO (SCREEN)**, but with the Y axis pointing up to preserve the X, Y and Z axis direction matching the default WORLD coordinate system.

NO (SCREEN CENTER) (`GLG_SCREEN_CENTER_COORD_SYSTEM` API constant)
Same as **NO (GLG SCREEN)**, but with the origin located in the center of the viewport.

When a new widget is created, the Resizable attribute of the widget's viewport is set automatically depending on the selected *Stretch/Resize* option used to create a new widget. For example, the GLG SCREEN setting is used for a widget created using the *File, New, Widget (Fixed Scale)* menu option.

Stretch

Controls mapping from view coordinates to window coordinates. If it is set to RESIZE, the original ratio of the viewport's height to width is not preserved and things may look distorted when the viewport is resized. If turned off (set to NO), the ratio is preserved. In this case *PushIn* attribute controls the rest of the mapping.The RESIZE AND ZOOM setting allows to stretch the viewport on both resizing and zooming. For more details about the mapping from view coordinates to window coordinates, see the *Coordinate Systems* section in *Structure of a GLG Drawing*.

PushIn

Controls which parts of the viewport are visible when Stretch is turned off. If *PushIn* is set to YES, the screen scaling factor is chosen in such a way that the viewport frame (a rectangle defined by two points with coordinates *(-1000,-1000)* and *(1000,1000)*) is completely visible in the window. Some other parts of the drawing may be visible as well depending on the screen ratio. If *PushIn* is set to NO, the screen scale factor is chosen as an average of the horizontal and vertical scaling factors defined by mapping the viewport frame to the window's border. Depending on the screen ratio, some parts of the viewport frame may be clipped off.

OpenGLHint

Controls the use of the OpenGL renderer for the screen's viewport. If set to OFF, a native GDI renderer is used, otherwise the OpenGL renderer is used if available. The flag is ignored in the Java and C#/.NET versions of the Toolkit.

The flag may have several *ON* values with different rendering priorities, from highest (1) to the lowest (3), which control the type of the OpenGL renderer to use: hardware or software. Viewports with higher priorities use hardware-accelerated renderer, while viewports with lower priorities may use software renderer. The software renderer may be used for icon buttons and other secondary windows with a small number of objects or infrequent updates. The use of the software renderer allows an application with a large number of viewports to use nice anti-aliased OpenGL rendering for all viewports without exceeding the graphics card's limit on the maximum number of OpenGL windows, which varies from card to card.

The runtime mapping of the OpenGL priorities to the type of the used OpenGL renderer is controlled by either command-line options, global configuration resources or environment variables. Refer to the *Hardware and Software Renderers, OpenGL Priority* section on page 20 for detailed description of the runtime mapping and all mapping options.

Even if the *OpenGLHint* is set to *ON*, the OpenGL renderer may be disabled by the *-glg-disable-opengl*

command-line option, setting the *GLG_OPENGL_MODE* environment variable to *False*, or setting the *GlgOpenGLMode* global configuration resource to 0. Refer to the *OpenGL or GDI (Native Windowing System) Renderer* section on page 18 for details.

OpenGL

Read-only attribute for the programming use that provides the current status of the OpenGL renderer. If the OpenGL renderer is successfully initialized and is used for rendering objects in the screen's viewport, the attribute will be set to the GLG_HARDWARE_OPENGL or GLG_SOFTWARE_OPENGL value, depending on the type of the OpenGL renderer used by the screen's viewport. If the GDI renderer is used, the attribute value will be set to GLG_NO_OPENGL.

ShellType

If this attribute is set to GLG_DIALOG_SHELL or GLG_APPLICATION_SHELL, the viewport appears in its own window, at the same level in the window hierarchy as the program operating it. The difference between the two is that the GLG_DIALOG_SHELL value always appears on top of other windows. The attribute may also be set to NONE (GLG_NO_TOP_SHELL), in which case, the viewport is simply a child window of a larger drawing.

WidgetType

The widget type may be selected from the list of possible widget types. For more information about widget types, see the *Native Widgets* chapter on page 214.

ShadowWidth

Defines the width of shadows. If the value differs from 0, the shadow bevels are drawn around the borders of the viewport. The sign of the *ShadowWidth* controls the type of the bevels: raised shadows for positive values and depressed shadows for negative values.

ExactColor

When using widget types besides Drawing Area on X Windows, this boolean attribute instructs the native widget to use the exact value specified by the *FillColor* attribute rather than take the closest available entry from the color table.

GridValue

Defines the grid spacing, in "world" coordinates. A viewport's corners are mapped to (-1000; -1000) and (1000; 1000) in this coordinate system. If it is zero, the grid is not drawn.

SpanX, SpanY

Define the mapping of the world coordinate system to the visible area of the viewport. The default value of the *SpanX* and *SpanY* attributes is 1000, which causes the corners of the visible area of the viewport to correspond to the coordinates (-1000, -1000) and (1000, 1000) in the absence of zooming and panning and with *Stretch* set to YES. The SpanX and SpanY attributes may used to change the mapping.

FonttableFile

Specifies a GLG drawing file containing a custom font table to use. A custom font table file can be shared between different viewports in multiple applications.

The drawing specified with *FonttableFile* may contain a font table saved using the *Save Fonttable* button in the font table object *Properties*. For the convenience of editing in the Builder, the drawing may also contain a *$Widget* viewport, in which case the viewport's font table will be used.

If a relative filename is used, it is interpreted relatively to the load path of the drawing first and then relatively to the current directory. For C/C++ and ActiveX deployment options, the GLG_PATH is also searched before the current directory.

The attribute may be set at run-time to customize fonts used in the drawing. **It needs to be set only on top-level viewports**, since children viewports inherit the font table from a parent. If a custom font table was defined using the *Add Font Table* button, it takes priority over the font table defined by the *FonttableFile* attribute.

Alternatively, an application can set a custom font table as a global Default Font Table by using the *GlgDefaultFontTableFile* and *GlgDefaultFontTable* global configuration resources described on page 345 of the *GLG Programming Reference Manual*. This custom font table will be inherited by all viewports that use a default font table.

FontTable

A custom font table that lists the fonts available for use in the viewport. If not defined, the font table defined by the *FonttableFile* is used. If the *FonttableFile* is not defined, the font table is inherited from the parent viewport, or the default GLG font table is used if no parent exists.

The *Add/Edit Font Table* button may be used to add a custom font table to the viewport or edit fonts in the font table if it already exists. To delete the font table, use the *Delete Table* button in the font table properties. Refer to the *Editing a Font Table* section on page 147 for information on editing a font table.

When a font table is added to all viewports in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added font tables (the attributes are constrained if the default *Constrained Clone* setting is used).

On Windows, the default font table uses the charset of the current system locale or the charset defined using the *GlgFontCharset* global configuration resource. In the X Windows environment, the default font table provides fonts with the ISO Latin1 extended ASCII character set (ISO 8859-1). A custom default font table may be supplied by an external file specified by the *GlgDefaultFontTabeFile* global configuration resource, see page 345 of the *GLG Programming Reference Manual.*

Colortable

The color table defines the colors available for use in a viewport. If not defined, the color table is inherited from the parent viewport, or the default GLG color table is used if there is no parent. On TrueColor systems with more than 256 colors (and also in Java and C#/.NET) the color table is not used for rendering: it is used only to define the number of colors in the Graphics Builder's color palette.

The *Add/Edit Color Table* button may be used to add a custom color table to the viewport or edit color table parameters if it already exists. To delete the color table, use the *Delete Table* button in the color table properties. See the *Colortable* section on page 144 for details.

When a color table is added to all viewports in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added color tables (the attributes are constrained if the default *Constrained Clone* setting is used).

Screen Transformation

The screen transformation is used by the screen object to adjust the drawing when the screen size changes. The parameters of the transformation are automatically set by the screen object, but they may be used for constraining purposes to obtain offsets in screen pixels. For example, the top part of the resource browser in the Builder has constraints that make it non-resizable, keeping a constant pixel size when the resource browser window is resized.

To obtain such an effect, a *MoveX* or *MoveY* transformation may be used, with the *Divide* transformation attached to its *Factor* attribute. The divisor of the *Divide* transformation is then constrained to the corresponding *X* or *Y Scale* of the screen transformation. This annuls the result of the screen size changes, making the move transformation maintain its offset in pixels rather then the world coordinates.

To get access to the parameters of the *Screen Transformation*, edit the attributes of the viewport object and press the *More* button to access the screen's attributes, then press *Edit Dynamics* to edit the screen's transformation.

Screen Name

This resource may be used at runtime to supply a string to be displayed as a window title of top-level viewports.

Advanced Graphical Objects

The GLG advanced graphical objects are structures with which one can build complex relations between simpler objects. Any three-dimensional object in a GLG drawing, for example, is represented as an agglomeration of simple, two-dimensional, shapes. The advanced objects also provide a drawing designer with tools to designate an object to be a template for other objects. This can be done for simple two-dimensional objects, as well as for complex compound objects.

Group

A group is a container object used to keep a set of simpler objects together. The group object does not have any control points nor any geometry. The geometry of the group is completely defined by the objects it contains. A group may contain any other objects, including other groups.

Though its most common use is to collect graphical objects into composite objects, the group is not really a graphical object. It can be used to collect data objects into lists and arrays. For example, a group is used to keep an array of *Custom Data Properties* attached to an object. The group object is included in this list because most users will encounter the group in trying to create composite graphical objects.

Groups may be used for creating layers, in which case the group's *Visibility* attribute may be used to control the visibility of its layer, rendering all objects in the group visible or invisible.

Aside from the name and other standard attributes (like *HasResources* and *Visibility*), the group object has only one attribute. The *DepthSort* attribute controls hidden surface removal for the child objects in that group. It operates the same as the *DepthSort* attribute of a viewport object (page 89), except that a group may inherit the value of this attribute from its parent object.

The following list describes the values the *DepthSort* attribute may have:

NO (GLG_NO)
Hidden surface removal is disabled

YES (GLG_YES)
Hidden surface removal is enabled.

If the GDI driver is used, a group is sorted as a whole. This means that if there are two intersecting groups, one group is completely in front of another.

SPECIAL (GLG_SPECIAL)
If the OpenGL driver is used: same as YES, activates hidden surface removal.

If the GDI driver is used: activates hidden surface removal using a faster but less precise depth sorting algorithm which uses objects' bounding boxes for determining the relative Z order of the objects.

INHERIT (GLG_INHERIT)
Inherit the value of the *DepthSort* attribute from the parent object. If the DepthSort attribute of some parent was set to YES and there were no intervening parents with the attribute set to NO, the inherited value of the attribute is YES, otherwise it is NO. The DepthSort attribute is not inherited across viewports, so the parent from which the YES value must be in the same viewport or the viewport itself.

PARTS (GLG_BY_PARENT)
If the OpenGL driver is used: Keep the current hidden surface removal state.

If the GDI driver is used: Activates hidden surface removal, but instead of sorting a group as a whole, elements of the group are sorted by a parent object with the *DepthSort* attribute set to a value different from NO. If there are two groups with the *DepthSort* attribute set to PARTS, and the *DepthSort* of the parent object containing these groups is set to YES or SPECIAL, the elements of these groups may be intermixed when drawn, based on their position in 3D space.

If a group's *DepthSort* is set to PARTS, the *Visibility* attribute of the group has no effect on the objects the group contains, since it is the group's parent who is drawing them. If you need to use the group's *Visibility* attribute to control the visibility of all objects in the group, constrain the *Visibility* attributes of all objects in the group to the group's *Visibility*.

In addition to the *DepthSort* attribute, the group's properties dialog also contains buttons for selecting and editing objects contained in the group, as well as adding and deleting objects from the group. The *EditAll* button starts editing the attributes of objects in the group by using the first object in the group to select a set of attributes for editing, which is a convenient option for fast editing of groups that contain objects of the same type.

For groups that contain objects of different types, the *Edit All (Select)* option allows you to select a set of attributes to edit. For example, if the group contains both the polygon and text objects, the *Edit All (Select)* option allows you to select the polygon or text attributes to be edited.

The rest of the buttons perform the same functions as the corresponding options of the *Arrange* menu, described on page 317.

Connector

The connector object may be used to connect other objects in the drawing. It is useful when implementing node and edge functionalities or connecting objects in a diagram.

There are two types of connectors. A recta-linear connector connects objects with linear segments, maintaining right angles between adjacent segments, and an arc connector connects objects with an arc path.

The connector has a set of **control points** which define its geometry. When the positions of its control points change, the connecting path changes as well, maintaining its recta-linear or arc shape. The end points of the connector can be constrained to the control points of other objects, so that the connector adjusts its geometry when the objects move. A *Reference* and *Container* objects may be used as containers to hold objects, providing a control point for constraining the connector's end points to. A Diagram Editor demo shows examples of using different connector objects to connect nodes in a diagram.

The recta-linear connector also has a set of **constrained points**. These points can't be moved, since their position is defined by the connector's control points, but they may be used to constrain other objects to them, staying attached to the middle point of the connecting path. When the connector is selected, either its control or constrained points are highlighted depending on the state of the *Options, Show Frame Points* option, which may be used to access the constrained points.

A connector object inherits polygon attributes (*EdgeColor*, *LineWidth* and *LineType*) but also has the following attributes:

EdgeType
A read-only attribute defining the object type of the graphical object used to render the connector. It is set to GLG_POLYGON (for recta-linear connectors) or GLG_ARC.

Direction
Defines the orientation of the first segment of a recta-linear connector, GLG_VERTICAL or GLG_HORIZONTAL.

PointList
A list of a connector's control points (recta-linear connectors only). Allows changing the order of points in the connector as well as adding and deleting them.

Series

The series object is used to produce multiple copies of the object defined as its **template**.

A series object has a variable number of control points. One point controls the location of the first instance's origin, while the others control the location of the line along which the series instances are distributed. An arbitrary transformation may be used to position the series' instances instead of

a linear path. For a straight-line series, there will thus be three control points: two to define the line, and one at the first object's origin. Note that when a linear series is created in the editor, the object's origin point is constrained to the first point of the series path.

There is another control point which is only visible while looking at the series template. This point controls the mapping of the template object onto the series constraint path. By default, this point is at the origin of the template object's coordinate system, so that the origin is mapped onto the path. It can be moved to another location to change mapping.

The series object has the following attributes:

Template
The object used as a template replicated in the series. It can be accessed for editing by traversing the hierarchy of the series object (*Traverse, Down* or the *Hierarchy Down* button ). When finished, use *Hierarchy Up* to return back to the top level. The *Template* resource is not visible in the *Resource Browser* by default, but will appear in the *Resource Browser* if it is named.

DepthSort
Controls hidden surface removal. This is the same as the *DepthSort* attribute for a group (page 98), except that the SPECIAL value uses a fast depth sorting algorithm optimized for Series objects when the GDI driver is used.

Factor
Defines a number of template copies to be created. Note that, strictly speaking, the *Factor* attribute of a series object defines not the number of copies produced, but the number of series object intervals. The actual number of copies may differ by one from the value of the *Factor* attribute. For example, for a vertical axis of a graph, the number of major ticks is greater than the factor by one, since an additional tick is placed at the end of the axis.
Every time the *Factor* attribute is changed, the old instances of the template are destroyed and the new number of instances is created. Any resource values set in the old instances are destroyed.

LogType
Defines how the template copies are positioned. If this attribute is set to NO, the copies are positioned linearly. If it is set to YES, they are positioned logarithmically. The base of the logarithm is equal to the value of the Factor attribute minus one.

Inversed
Controls how the instances are named. If the value is DIRECT, instance 0 is at the start of the series path. If the value is INVERSED, instance 0 is at the end.

CloneType
The clone type used to create instances of the template. It may be *Full, Weak, Strong or Constrained*. Refer to the *Cloning an Object* section of the *Using the GLG Graphics Builder* for details on using the clone type.

Persistent
Controls persistency of attribute settings for the series' instances. If set to VOLATILE (default), only the series' template will be saved, and the instances will be created from the template on hierarchy setup performed at loading time. Any settings of the local attributes of the instances will be lost.
If the PERSISTENT setting is used, the instances will be saved in the drawing, preserving current settings when the drawing is loaded. If the number of instances is increased by

changing the series' *Factor*, the additional instances will be created by copying the template. The existing instances and their attribute settings will be preserved, which may be used for setting line attributes of multi-line and other multi-set graphs in the Builder.

Recreate Instances

This button is present only in the Builder and may be used to discard the instances by recreating them from the template. To discard the instances at runtime, set the series' *Factor* to zero and call *GlgUpdate*.

PathXform

PathXform defines the shape of the path used by the series. The type of the PathXform transformation (*Move*, *Rotate*, etc.) is defined when the series object is created. The path transformation's parameters may be edited at any time.

Instances of the template are created by the series after the drawing hierarchy has been setup. To differentiate the instances, a zero-based index is added as a suffix to the name of a template to construct the name of an individual copy. For example, if the name of a template is *Label*, the produced copies of it have names *Label0*, *Label1*, *Label2*, and so on. Each of these names appears in the resource hierarchy, as does the name of the original, providing access to both the template and its instances. The instances may be accessed only after they've been created (after the drawing hierarchy has been setup).

The *Global* flag of the template's attributes may be used to constrain attributes of its instances. If an attribute's *Global* flag is set to GLOBAL in the template, changing this attribute will change all instances. If the flag is set to LOCAL, the attribute of each instance may be set independently, enabling independent dynamics. The constraining behavior of the series is also controlled by its *CloneType* attribute.

When the series object is used in GLG widgets, it may scale instances of the template. For example, if you increase the number of bars in a GLG bar graph widget, the bars become narrower so they will fit on the X axis. Series with this feature cannot be created in the GLG Graphics Builder (you can still use the square series, which scale their instances, to obtain a similar effect). However, you can use the Builder to edit graphs that use it. For a series with this feature enabled, a rectangle defined by points with coordinates *(-1000,-1000)* and *(1000,1000)* in the template object coordinate system is mapped to the size of one bar slot.

Square Series

The square series object is a special case of a series object used to position copies of a template object on a two-dimensional grid.

Like a parallelogram, the square series object has three control points that define a corner and two sides. These three points define a parallelogram. The first defined point is the center. Rows of the square series are parallel to the line through the center and the second defined point, and columns are parallel to the line through the center and the third defined point.

Created copies are scaled to fit the area outlined by the parallelogram. The viewport frame of the template is mapped to the size of one slot of the square series.

The square series object has the following attributes:

Template
The object used as a template replicated in the series. It can be accessed for editing by traversing the hierarchy of the series object (*Traverse, Down* or the *Hierarchy Down* button ). When finished, use *Hierarchy Up* to return back to the top level. The *Template* resource is not visible in the *Resource Browser* by default, but will appear in the *Resource Browser* if it is named.

DepthSort
Controls hidden surface removal. Similar to the *DepthSort* attribute of a group (page 98), except that the SPECIAL value uses a fast depth sorting algorithm optimized for series objects when the GDI driver is used.

RowFactor
Defines a number of rows of template instances.

ColumnFactor
Defines a number of columns of template instances.

ColumnsFirst
Controls how instances of the template are numbered. If set to YES, they are numbered by the columns, or by the rows otherwise.

CloneType
The clone type used to create instances of the template. It may be *Full, Weak, Strong or Constrained*. Refer to the *Cloning an Object* section of the *Using the GLG Graphics Builder* for details on using the clone type.

KeepEditRatio
If set to YES, preserves the X/Y ratio of the template drawing while editing its content in the Builder by going down into it using the *Hierarchy Down* button.

Persistent
Controls persistency of attribute settings for the series' instances. If set to VOLATILE (default), only the series' template will be saved, and the instances will be created from the template on hierarchy setup performed at loading time. Any settings of the local attributes of the instances will be lost.
If the PERSISTENT setting is used, the instances will be saved in the drawing, preserving current settings when the drawing is loaded. If the number of instances is increased by changing the series' row and column factors, the additional instances will be created by copying the template. The existing instances and their attribute settings will be preserved.
Keep in mind that the instances are numbered sequentially, and the row and column of the preserved instances may change if the *ColumnsFirst* attribute or the series' row or column factors are modified.

Recreate Instances
This button is present only in the Builder and may be used to discard the instances by recreating them from the template. To discard the instances at runtime, set the series row or column factor to zero and call *GlgUpdate*.

Like a one-dimensional series, a square series has a template object that is replicated to form the series. As with the series, created instances of the template are differentiated by the zero-based index added to the name of the template. Instances are numbered using one sequential index, to refrain from using two separate row and column indexes.

The *Global* flag of the template's attributes may be used to constrain attributes of its instances. If an attribute's *Global* flag is set to GLOBAL in the template, changing this attribute will change all instances. If the flag is set to LOCAL, the attribute of each instance may be set independently, enabling independent dynamics. The constraining behavior of the square series is also controlled by its *CloneType* attribute.

Reference

A *Reference* object is a wrapper around a group of objects used as a "template". The *Reference* object may be used to replicate the same template in multiple places in one drawing or in multiple drawings. It may also be used as a convenient wrapper for positioning a group of object using a single anchor point. There are three flavors of the *Reference* object used to implement different functionality:

- ***Container*** encapsulates a collection of objects in a single entity. Unlike the group, the container also provides a single control point for positioning it in the drawing. A container's *Template* holds the objects drawn in the container. If the container is copied, the contained template object is copied as well, so that each copy of the container has its own independent template. A container draws its template directly, without creating any additional instances of it.

  Containers may be used to implement **node/edge** functionalities. If a container is used as a node, it's template's control points are protected and the container's single control point may be conveniently used for positioning or attaching connectors to. Containers may also be used to preserve center of rotational dynamics when objects are moved.

- ***SubDrawing*** is used to replicate a single shared template in different locations in one drawing or in multiple drawings. The subdrawing object has a single control point that defines its position. All subdrawing instances can be changed in one place by editing the subdrawing's template. This is useful when constructing drawings that contain many copies of the same object that needs to be edited in one place. The template may be included in the drawing or stored in an external file. The *Source* parameter of the subdrawing defines where the template is stored.

  At runtime, attributes of each *SubDrawing* instance may be changed independently or made global for all instances. This is controlled by setting a *Global* flag of a particular attribute in the template. If the flag is set to LOCAL, the attribute may be changed independently for each instance at runtime. If it is set to GLOBAL, changing the attribute of any instance (or the template) will affect all of them. For example, if the template has a resource *LabelColor* which is GLOBAL, changing this resource for one instance will affect all subdrawing instances in the drawing.

  Any changes to the attributes of a subdrawing instance are volatile and not saved with the drawing. When the drawing is loaded, each subdrawing instance is created by copying the subdrawing's template, and all attributes of an instance are initialized to the values of the corresponding template attributes. ***Bindings*** may be used to make some attributes persistent and to specify unique attribute values for each instance of the subdrawing. An attribute is made persistent by setting its *Global* flag to BOUND in the template. Any changes to the BOUND attributes of a subdrawing instance are saved with the drawing. Refer to the *Bindings* section on page 108 for more information.

A subdrawing may be used for icons that change their shapes depending on the icon type. For example, an icon representing an airplane can display different shapes depending on the airplane type. This functionality may be achieved using **subdrawing dynamics** or **object dynamics**. Subdrawing dynamics changes the drawing file used as a template to display a different drawing. Object dynamics uses a single template that contains several objects and provides a mechanism for selecting the object to be displayed. This achieves the same effect as subdrawing dynamics without loading a different drawing file. Refer to the description of the *Reference* object attributes below for further details.

Subdrawings may use a template stored in an external file or included in the drawing. Like a container, a subdrawing may be used as a node that provides a single control point for positioning or attaching connectors to it.

By using subdrawings, you can make a drawing file smaller than it might be otherwise, since only one copy of the template is saved. The drawback of using subdrawings is that the initial resource settings for instances of the template can not be saved in the drawing and must be done programmatically at run time. This limitation can be avoided by using **attribute binding** described later in this section on page 108.

- ***SubWindow*** 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* may be used as a subdrawing with two control points, which is useful for interface objects such as buttons, icons 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. Refer to the *Bindings* section on page 108 for more information.

A *Reference* object has the following attributes:

Template

The original template object shared by all copies. For file and palette references, this resource is NULL until the drawing hierarchy is set up. After the setup, the template contains the template object loaded from a file or palette.

To edit the template of any reference object (*Container, SubDrawing* or *SubWindow*), use *Traverse*, *Hierarchy Down* (or). If the template is stored in a separate drawing file, traversing down loads the template drawing. If the template uses a palette, traversing down traverses down the palette object. When finished, use *Traverse*, *Up* (or)to return back to the top level.

The options in the *Options, Subdrawing Traversal* menu control the Builder's verbosity settings when returning to the subdrawing level after editing the subdrawing's template. The default *Verbose* option presents a prompt before saving the modified template drawing, providing the user with options for saving the drawing in a different file or discarding the changes. The *Silent (Auto-Save)* option automatically saves the modified drawing into the same file without displaying a prompt.

When a template is modified, the changes take effect when the drawing containing subdrawings is loaded and set up, causing subdrawings to create their *Instances* from the *Template*. In the GLG Builder, it happens automatically when the drawing is reloaded on *Hierarchy Up* after the template editing is finished. If the template drawing in an external file that was changed outside of the GLG Builder, use the *Reset*  toolbar button to update the drawing. An application can reset drawing hierarchy at runtime to reload the template.

Instance

For the *SubDrawing* and *SubWindow* objects, contains a local copy of the template used for rendering. If the *ObjectPath* attribute is not NULL, the Instance contains not the whole template, but its subobject used for rendering as defined by the *ObjectPath*. The *Instance* is created dynamically after the drawing hierarchy is set up. A user may edit its resources, but the changes are volatile: when the drawing is reloaded or reset, the attributes of the instance are recreated from the *Template* again, erasing any modifications a user might have made (except for *Global* attributes that are shared between all instances, and rebound attributes described in the *Bindings* section on page 108).

The *Instance* object is visible in the *Resource Browser* as a resource. For example, if the template object is named *MyTemplate*, both "*Instance*" and "*MyTemplate*" resources will be visible and refer to the same local copy, while the "*Template*" resource points to the original template shared by all copies.

For the *Container* object, the template is displayed directly without creating a local copy, and both *Instance* and *Template* resources refer to the same object.

ReferenceType

The subtype of a reference object: GLG_CONTAINER_REF, GLG_SUBWINDOW_REF or GLG_SUBDRAWING_REF. This is a read-only attribute that is set at the creation time and defines the reference as either a *Container*, *SubWindow* or *SubDrawing*.

Source

The source of the template object for the *SubDrawing* and *SubWindow* objects. Possible values are:

GLG_USE_FILE - uses a template stored in an external drawing file specified by the *SourcePath* attribute.

GLG_USE_INCLUDED - uses a template included in the drawing. The template is stored as a part of the subdrawing. When the subdrawing is copied in the same drawing, the template is shared between all copies. The template is not shared between subdrawings in different drawings.

GLG_USE_PALETTE - a special case of the included template that allows embedding a palette of objects in the drawing for easy editing. The palette object is specified by the *SourcePath* attribute, which defines the palette's resource path in the drawing. If the *SourcePath* is not set, *"$Palette"* is used as a default palette path, which assumes an object named $*Palette* at the top level of the drawing hierarchy.

An example of a palette is a viewport object containing several graphical objects used as icons in subdrawings. Using a palette makes it easier to locate the template's icons when they need to be edited.

The *Source* attribute is set at the creation time depending on the way the object was created, such as *SubDrawing From File* or *SubDrawing From Object*. If the object is created using *SubDrawing From File*, the value of its *Source* may be changed from GLG_USE_FILE to GLG_USE_INCLUDED to permanently store the loaded template in the subdrawing, as opposed to using an external file.

The *Source* attributes of subdrawings may be constrained, making it possible to change the template storage type of all subdrawings (from the referencing a separate file to including a template object in the drawing and back) by setting the *Source* attribute of a single *SubDrawing* object.

SourcePath

For **subdrawings that use a template stored in an external file (*Source=File*)**, the attribute defines the location of the drawing file containing the template. When the file is loaded, the object named *$Drawing* or *$Widget* is used as the template object. If neither named object is found, the whole drawing is used as a template.

Note: When a subdrawing is created in the Graphics Builder, a file browser is used to select the subdrawing file, and an absolute path is stored in *SourcePath*. To allow the application to be moved to a different directory or a different environment (web or Java) without adjusting subdrawing paths, it is recommended to edit stored *SourcePath* to make it relative to the location of the drawing.

If *SourcePath* defines a relative file name, the system tries to resolve it in the following order:

- attempting to load the file relative to the directory of the drawing
- trying to locate the file in one of the directories defined by the *GLG_PATH* environment variable or the *GlgSearchPath* global configuration resource
- attempting to load the file relative to the current directory as the last resort.

Cross-Platform Use Note: For cross-platform, Java and web-based deployment, use '/' as a path delimiter even on Windows. On Windows, the Builder converts '/' to '\' automatically when necessary.

To implement **subdrawing dynamics**, a *List* transformation may be attached to the *SourcePath* to specify a list of drawing files. If the index of the List transformation changes, a different template drawing will be displayed. The *ObjectPath* attribute described below may be used for even more efficient **object dynamics**.

For **subdrawings that use a palette (*Source=Palette*)**, the attribute defines a resource path for accessing the palette inside the drawing. The palette is usually a viewport that contains a collection of objects used for object dynamics. If the *SourcePath* is not set, the default *$Palette* resource name is used. If an application loads the drawing using a *LoadWidget* method, the resource path must be relative to the *$Widget* viewport.

ObjectPath

Defines an object to be displayed in the subdrawing by specifying a resource path of the object within the template. If *ObjectPath* is set and points to an object inside the template, only that object will be displayed in the subdrawing. The template may have several named objects, and changing *ObjectPath* to a different object name will display a different object. The *ObjectPath* is relative to the template. If *ObjectPath* is not set, the whole template will be displayed.

For subdrawings that use an external file (*Source=File*), the template is loaded from a subdrawing file; if the file contains an object named "*$Drawing*" or "*$Widget*", *ObjectPath* is relative to that object, otherwise it is relative to the whole template drawing.

The *ObjectPath* attribute makes it possible to implement subdrawing dynamics using just one template containing several objects, instead of placing each object into a different

drawing file and loading a new template drawing every time the subdrawing changes its shape. In other words, the subdrawing dynamics implemented via the *SourcePath* attribute may now be replaced with more efficient **object dynamics** via the use of *ObjectPath*. The the object dynamics also works with included subdrawings.

To implement object dynamics, a list transformation is attached to the *ObjectPath* attribute to define a list of resource paths pointing to different objects in the template. When the index of the list transformation changes, a different object path is used and a different object from the template is displayed in the subdrawing. Each value in the list may include an anchor path as described below.

The *ObjectPath* attribute may also contain two resource paths separated by a colon. The second path specifies the **anchor path**: a resource path to a control point to be used as an anchor when displaying the object. The anchor path may point to a named control point of an object, or to a control point of a marker outside of the object used to control object's position. For example, a template may contain a polygon object named *Triangle* with *HasResources=YES* and one of it's control points named *Anchor*. The *Triangle:Triangle/Anchor* setting of *ObjectPath* will display the polygon anchored at the *Anchor* point. The template may also contain an arc object named *Arc* and a marker object named *ArcAnchor* used to control the arc's display position. The *Arc:ArcAnchor/Point* setting may be used to display the arc anchored at the position of the *ArcAnchor* marker (the default *Point* attribute name is used to reference marker's point). The anchor values may be accessed via the *CoordOrigin* resource of the *SubDrawing* object. An object's anchoring is also affected by the value of the subdrawing's *Origin* attribute described below. Set *Origin* to (0;0;0) to anchor the subdrawing at the exact position of the anchor object specified by the anchor path.

If *ObjectPath* is not set (or if the portion of the string before the colon separator is empty), the whole template object will be displayed.

CloneType

The clone type used to create an instance when copying the template. It may be *Full, Weak, Strong or Constrained*. Refer to the *Cloning an Object* chapter on page 258 for details on using the clone type. The default value is STRONG clone, which constrains attributes with GLOBAL and SEMI_GLOBAL settings of the *Global* flag. *CloneType* has no effect if *EnableCache* is set to NO.

FixedSize

Determines if a *SubDrawing* or *Container* object can be resized. If set to YES, the object does not resize when the drawing is resized or zoomed in, otherwise it is resized with the drawing. The size of a *SubDrawing* or *Container* object of a fixed size may be changed only by editing its template.

If the *FixedSize* is set to YES, the point coordinates of the object's template are interpreted as GLG screen coordinates. If the *FixedSize* is set to NO, the point coordinates of the object's template are interpreted as world coordinates of the drawing. Subdrawing and container objects are created with the *FixedSize* initially set to NO, and the template's coordinates are interpreted in the world coordinate space. When the *FixedSize* is changed to YES, the visible size of the object changes since the template is now drawn in the screen coordinate space. The size of the fixed size subdrawings and containers may be adjusted by traversing down into the object's template using the *Hierarchy Down* button and changing the template's size.

EnableCache

Enables or disables template cache for subdrawings and subwindows that use template stored in an external file. If set to YES, 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, each subdrawing loads its own copy of the template, and the *CloneType* attribute has no effect (attributes are not constrained). *EnableCache* may be set to NO to increase performance for *SubWindows* that are used to 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.

EnableCache has no effect for subdrawings and subwindows that use included or palette template.

KeepEditRatio

If set to YES, preserves the X/Y ratio of the template drawing while editing its content in the Builder by going down into it using the *Hierarchy Down* button. If set to NO, the template drawing may appear stretched.

Bindings

A slot for attaching an optional array of rebound attributes (bindings) which enable the user to define local settings for instances of subdrawings and subwindows. Resource settings of bound attributes are persistent and are saved with the drawing, which allows the user to customize each instance by assigning unique resource settings in the GLG Builder. For example, bindings may be used to assign unique data tags to individual subdrawing instances to animate them with data from different sources.

SubDrawing and *SubWindow* objects instantiate a copy of their template, which means that all instances of a SubDrawings or SubWindow with the same template will use the same initial attribute values taken from the template. For example, if a subdrawing represents a tank filled with liquid, and the drawing contains several such tanks, the initial liquid color for all tanks will be the same when the drawing is loaded. **Bindings** may be used to define a different liquid color for each tank that overrides the color defined in the template. Bindings may also be used to constrain a color in the subdrawing to the color of another object in the drawing. For example, the color of the tank may be constrained to the color of pipes connected to it.

Bindings for an attribute, such as the liquid color in the previous example, are activated by naming the attribute in the template and setting its *Global* flag to BOUND. If an attribute is BOUND, the attribute is stored in the drawing, instead of using the attribute from the subdrawing's template. When the subdrawing is loaded, the attribute from the template is replaced with (bound to) the attribute stored in the drawing.

When a subdrawing with named BOUND attributes is placed in a drawing, it will have a *Bindings* array containing local copies of the bound attributes. These attributes may be edited to define unique settings for each subdrawing instance. Bindings are saved with the subdrawing. When a subdrawing is loaded, all named bound attributes of its *Instance* will be "rebound", i.e. replaced with the corresponding local copies stored in the *Bindings* array.

The local copies of the attributes in the *Bindings* array may be edited using the *Edit*

Bindings button in the subdrawing's *Properties* dialog. The user can change the values of the attributes and add unique tags to each of the subdrawing instances. The user can also change the names of the rebound attributes, constrain them to attributes of other objects in the drawing or add dynamics to them to modify behavior of the subdrawing's instance. These changes will affect only the instance of the subdrawing the *Bindings* are attached to. If a name, tag name or tag source of a rebound attribute is changed, only the new values will be visible in the subdrawing's *Instance*, while the old resource name and tag will still be visible in its *Template*.

To reset bindings, press the *Reset Bindings* button. This discards the old bindings and recreates the *Bindings* array with the initial values copied from the template, so they may edited from scratch.

If an application uses several levels of nested subdrawings (or subwindows), a subdrawing may be used as a template by its parent subdrawing. When a BOUND attribute is rebound to a local copy stored in the *Bindings* array, the *Global* flag of the local copy is reset to LOCAL by default, which disables further rebinding by any parent subdrawings or subwindows. To enable attribute rebinding across several levels of nested subdrawings, the *Global* flag of the local copy in the *Bindings* array can be set to BOUND again to enable rebinding by the parent subdrawing.

Origin

The geometrical attribute that controls anchoring of the template for *SubDrawing* and *Container* objects. The *Origin* attribute is not shown in the object's *Properties* dialog, but is visible as a resource in the *Resource Browser*. For containers and subdrawings with an included template, its position is also displayed as a round marker when the object's template is edited using the *Hierarchy Down* button (the marker may be hidden behind the template's objects).

For *Containers* and *SubDrawings* with no anchoring defined in the *ObjectPath*, the template will be anchored at a position defined by the *Origin*: the control point of the subdrawing will be at the same location as indicated by the Origin marker in the *Template*. For example, if *Origin* is set to (100;0;0), the subdrawing's single control point will coincide with the (100;0;0) position in the template. When a *Container* or a *SubDrawing* is created, the value of the attribute is set based on the user input. The value may be changed or reset to *(0,0,0)* by setting the *Origin* resource, moving the *Origin* marker in the template, or by dragging the subdrawing's anchor point using *Ctrl-Shift-Left-Mouse-Move*.

If an additional anchor path is specified in the *ObjectPath* attribute, the template is further offset by the distance from the center of the template's coordinate system to the anchor object defined in the anchor path of the *ObjectPath* attribute. Another way of looking at it is that the origin's anchoring offset is applied to all objects in the template on top of the individual object's anchoring defined in the *ObjectPath*.

A special feature of the *SubDrawing* objects is that any number of copies of the object may be made, with all copies using the same template object. Thus, subdrawings become useful in any drawing where there are a large number of identical (or nearly identical) objects. Editing the template will affect all instances of it in the drawing. If a subdrawing uses a template stored in an external file, all instances of the subdrawing may be changed by editing its template file. Rebinding may be used to modify the local copy of the template as described in the *Rebindings* section above.

The *Global* flag of a template's attributes may be used to constrain attributes of individual instances. If an attribute's *Global* flag is set to GLOBAL in the template, changing this attribute will change all instances in the drawing. If the flag is set to LOCAL, the attribute of each instance may be set independently, allowing for independent dynamics. The constraining behavior of the reference object is also controlled by its *CloneType* attribute. If the *Global* flag is set to BOUND, the attribute is rebound and its value is taken from the rebinding table, as described above. Rebound attributes must be named.

When a container is copied, each copy will have its own independent copy of the template.

If *SubDrawing* or *Container* objects are used to represent connected nodes, the objects' single control point may be used to constrain connecting lines that represent edges.

Polyline

The polyline object is a special type of series that produces an open polygon with specified number of points or a specified number of line segments. It is often used for animated line graphs like the ones in the GLG widget set.

A polyline has a variable number of control points defining its location. It also has the following attributes:

Factor
Defines number of polyline control points to be created.

DrawMarkers
Controls the presence of polyline's markers. Markers are created if the attribute is set to YES.

DrawLines
Controls the presence of the lines between a polyline's points. These lines are drawn between each marker if the attribute is set to YES.

Segments
Enables the segments mode. If the value set to YES, separate polygon objects are used to render each segment of a polyline, allowing to use different colors and line attributes for each segment. In this case the *Polygon* attribute is used as a template for producing the segment instances. If the value is set to NO, one polygon is used, providing faster and more efficient rendering.

Marker
A template marker object. Copies of this marker appear on each control point if the *DrawMarkers* attribute is turned on. To access the marker template, traverse the hierarchy of the polyline (the *Hierarchy Down* button ). When finished, use *Traverse*, *Up* to return back to the top level.

Polygon
A template polygon object. The lines connecting polyline points inherit their graphical properties from this template. Therefore, you can modify the attributes of this polygon to change the line color and width. If *Segments* is set to YES, the polygon is used as a template for creating the segments. If *Segments* is set to NO, the polygon itself is used to render the polyline.

CloneType
The clone type used to create an instance of the polygon and marker templates. It may be *Full, Weak, Strong or Constrained*. Refer to the *Cloning an Object* section of the *Using the GLG Graphics Builder* for details on using the clone type.

In a manner similar to the series object, instances of markers and polyline segments are differentiated by the zero-based index added to the name of the corresponding template object. For example, for a three-point polyline, the resource hierarchy might include the template objects *Mark* and *Poly*, and the series instances *Mark0*, *Mark1*, and *Mark2*, and *Poly0* and *Poly1*.

If markers are named, the polyline contains a group named *Markers* which in turn contains the instances of the marker template. If the template marker's conrol points are named, the polyline also contains a group called *Points*, containing the instances of the control points. If the template polygon is named, there is a group called *Polygons*, containing those instances, too.

The *Global* flag of a marker template's attributes may be used to constrain attributes of the marker's instances. If an attribute's *Global* flag is set to GLOBAL in the template, changing this attribute will change all instances. If the flag is set to LOCAL, the attribute of each instance may be set independently, allowing for independent dynamics. When the *Segments* mode is enabled, the GLOBAL flag of the polygon template's attributes may be used to constrain attributes of the segment polygons in the same way. The constraining behavior of the polyline is also controlled by its *CloneType* attribute.

Polysurface

The polysurface object is a special type of series object used to create a surface made of a number of surface patches. The number of patches is determined by the row and column factors of the polysurface. A patch is a polygon connecting four neighboring points of a surface. The polysurface is used in several three-dimensional graphs in the GLG Widget Set.

In addition to the three control points defining the boundary parallelogram, the polysurface also has the following attributes:

DepthSort
Controls hidden surface removal. Similar to the *DepthSort* attribute of a group (page 98), except that the SPECIAL value uses a fast depth sorting algorithm optimized for polysurface objects when the GDI driver is used.

RowFactor and **ColumnFactor**
Define a number of polygon patches used to form the surface.

DrawMarkers
Controls the presence of markers at the vertices of the polysurface. Markers are drawn only if the attribute is set to YES. Markers always appear on top of the surface patches.

Polygon
A template defining attributes of the polygon patches.

Marker
A template marker object defining attributes of the markers.To access the marker template, traverse the hierarchy of the polysurface (the *Hierarchy Down* button ). When finished, use *Traverse*, *Up* to return back to the top level.

CloneType
The clone type used to create an instance of the polygon and marker templates. It may be *Full, Weak, Strong or Constrained*. Refer to the *Cloning an Object* section of the *Using the GLG Graphics Builder* for details on using the clone type.

The naming of instances of template markers and polygon patches is analogous to the naming of the comparable objects for a polyline. See page 111.

The *Global* flag of a marker template's attributes may be used to constrain attributes of the marker's instances. If an attribute's *Global* flag is set to GLOBAL in the template, changing this attribute will change all instances. If the flag is set to LOCAL, the attribute of each instance may be set independently, allowing for independent dynamics. The GLOBAL flag of the polygon template's attributes may be used to constrain attributes of the polygon patches in the same way. The constraining behavior of the polysurface is also controlled by its *CloneType* attribute.

Frame

The frame object is used to constrain the geometry of other objects. A frame has a number of **control points** defining its own geometry and position, and an array of **constrained points (frame points)** used for constraining the geometry of other objects. When the frame is selected, either its control or frame points are highlighted depending on the state of *Options, Show Frame Points* option.

Frame constrained points can not be edited independently, since their position is determined by the frame's control points. The control points may be moved to change the position of the frame points, but the frame points themselves cannot be moved directly. Anything constrained to the frame points moves with the frame. To access the constrained points, use *Options, Show Frame Points*; see page 358.

There are several types of frames:

1D Frame
has frame points positioned along the line defined by two control points.

2D Frame
has frame points positioned inside the parallelogram defined by three control points.

3D Frame
has frame points positioned inside the parallel prism defined by four control points.

Free Frame
is simply a collection of arbitrarily positioned frame points. A **point frame** is the special case of a free frame with one point.

Except for the free frame, the number of frame points is indirectly defined by the frame's factor. The factor defines a number of intervals between the frame's frame points along every frame's axis.

A frame object has the following read-only attributes:

FrameType
Defines the number of frame's dimensions and can be 1D, 2D, 3D, or FREE

FrameFactor
Defines the number of intervals between frame points along each frame's axis. The number of frame points is one more than the number of intervals.

Chart Objects

Chart

The *Chart* object is used to render real-time charts with large number of data points and fast update rates. A real time chart is highly optimized to handle hundreds of thousands of data points and update the display hundreds times per second. The chart supports integrated zooming and scrolling, chart tooltips, cursor feedback and data sample selection.

A chart object has two control points that define the geometry of its data area. The chart object is a composite object that contains a number of subcomponents, such as plots, axes and other auxiliary objects. The chart creates a required number of plots and axes and manages their layout. The chart object also handles chart tooltips, point selection and cross-hair cursor. The *AxisType* attribute of the chart's X axis defines the type of the chart: a scrolling strip-chart or an XY scatter.

A chart object has the following attributes:

Number of Plots
Specifies a number of plots in the chart. The chart automatically adds or deletes plots when the value of the attribute is changed. When plots are added or deleted via the *GlgAddPlot* or *GlgDeletePlot* API methods, the chart adjusts the value of the attribute to match the new number of plots.

When new plots are added, they are assigned default names formed by appending the plot's index at the end of the *"Plot#"* string, i.e. *"Plot#0"*, *"Plot#1"* and so on. When plots are reordered, default names change to reflect the new order; a persistent custom name can be assigned to each plot.

Number of Y Axes
Specifies a number of Y axes in the chart. The chart automatically adds or deletes Y axes when the value of the attribute is changed.

When new Y axes are added, they are assigned default names formed by appending the axis's index at the end of the *"YAxis#"* string, i.e. *"YAxis#0"*, *"YAxis#1"* and so on. When axes are reordered, default names change to reflect the new order; a persistent custom name can be assigned to each Y axis.

Number of Levels
Specifies a number of level lines in the chart. A level line object annotates a chart threshold by drawing a horizontal line at a specified height. The chart automatically adds or deletes level lines when the value of the attribute is changed.

When new level lines are added, they are assigned default names formed by appending the level' index at the end of the *"Level#"* string, i.e. *"Level#0"*, *"Level#1"* and so on. When level lines are reordered, default names change to reflect the new order; a persistent custom name can be assigned to each level line.

Common Range
Controls whether or not the chart ranges are shared across the elements of the chart. If set to YES, all plots and level lines in the chart, as well as the first Y axis, share the same Low/High range. If set to NO, each plot and level line may have a different set of High/Low ranges.

Auto Scroll (scrolling charts only)
Controls the automatic scrolling mode. If set to YES, every time a new sample is pushed into the chart, the chart will automatically scroll to accommodate the new sample at the end of the chart. If set to NO, the chart can be scrolled as needed by changing *EndValue* of its X axis.

Draw Grid
Enables or disables the X and Y grid lines. Possible values are NONE, X, Y and XY.

Edit Plots
A button in the *Properties* dialog for editing the chart's plots. It activates the *Plot List* dialog as well as the *Properties* dialog for the selected plot. A plot's *Opacity* attribute can be set to 1or 0 to turn individual plots on or off without losing accumulated samples.

The *Plot List* dialog has buttons for adding, deleting and reordering plots in the list. When plots are reordered, plots with default names (such as *"Plot#0"*, *"Plot#1"*, etc.) are assigned new default names that match their new position in the list. A persistent custom name can be assigned to each plot.

Edit X Axis
A button in the *Properties* dialog for editing properties of the chart's X axis. The *AxisType* attribute of the X axis determines the scrolling behavior of the chart: setting it to RANGE changes the chart type from a scrolling chart to the XY Scatter chart. The axis's *Visibility* attribute can be used to turn the X axis on or off.

Edit Y Axes
A button in the *Properties* dialog for editing the chart's Y axes. It activates the *Y Axis List* dialog as well as the *Properties* dialog for the selected Y axis. An axis's *Visibility* attribute can be used to turn individual Y axis on or off.

The *Y Axis List* dialog has buttons for adding, deleting and reordering axes in the list. When axes are reordered, axes with default names (such as *"YAxis#0"*, *"YAxis#1"*, etc.) are assigned new default names that match their new position in the list. A persistent custom name can be assigned to each Y axis.

Edit Background
A button in the *Properties* dialog for editing attributes of the chart's drawing area. To disable the background, set its *FillType* to NONE or set its *Opacity* to 0.

Edit Grid
A button in the *Properties* dialog for editing attributes of the chart's X and Y grid lines. Use the chart's *DrawGrid* attribute to enable or disable the grid.

Edit Level Lines
A button in the *Properties* dialog for editing the chart's level lines. It activates the *Level Line List* dialog as well as the *Properties* dialog for the selected level line. A level's ' *Opacity* attribute can be set to 1 or 0 to turn individual levels on or off.

The *Level Line List* dialog has buttons for adding, deleting and reordering axes in the list.

When level lines are reordered, level lines with default names (such as *"Level#0"*, *"Level#1"*, etc.) are assigned new default names that match their new position in the list. A persistent custom name can be assigned to each level line.

Sort Input

If set to YES, the chart will find the proper insertion place for the new sample based on its time stamp to ensure that all time stamps are in the increasing order. It may be useful when data samples have time stamps, but some samples may be delayed in transmission and arrive out of sequence. If set to NO, the chart will always insert the new sample at the end.

Buffer X Span (scrollable charts only)

Controls the time span (in seconds) of the chart's data history buffer and is used to define a data buffer larger than the chart's visible span on the X axis. For example, if *Span* of a scrolling time chart is set to 600 and its *BufferXSpan* is set to 3600, the chart will show data for the last 10 minutes but will keep all samples for the last hour in the data buffer, allowing the user to scroll the chart. Samples older than one hour will be automatically discarded when the new samples come in.

If *BufferXSpan* is set to 0 (default), the chart will maintain all samples visible in the chart's *Span* and will discard samples which scroll out and become invisible. This is equivalent to *BufferXSpan* having a value equal to the value of the chart's *Span* attribute.

If *BufferXSpan* is set to -1, the chart will not discard samples based on their time stamp. The number of samples kept in the chart's buffer can still be controlled by the chart's *BufferSize* attribute.

The value of the attribute is ignored for non-scrollable XY Scatter charts. For non-time based scrolling charts, *BufferXSpan* is defined in the units of the X axis.

Buffer Size

Controls the maximum number of samples per plot stored in the chart's data history buffer. When new samples are pushed into the chart and the size of the data buffer is exceeded, the oldest samples are discarded to maintain the requested maximum number of samples in the buffer.

The *BufferSize* may be used to limit the size of the data buffer with or without *BufferXSpan*. For example, if the *BufferXSpan* is set to 3600 to keep all samples for the last hour, the number of data samples in the buffer may exceed 3600 if new samples are pushed into the chart faster than once per second. *BufferSize* may be set to, let's say, 7200 to limit the maximum number of samples kept in the buffer.

If *BufferSize* is set to 0, the chart will not limit the maximum number of samples in the data buffer, and the number of samples in the buffer will be limited only by the settings of the *BufferXSpan* attribute. If *BufferSize* set to 0 and *BufferXSpan* is set to -1, the data buffer will accumulate an unlimited number of samples, which can cause an application to run out of memory and should be used with caution.

The samples accumulated in the chart's data buffer are discarded when the chart is reset. To discard the samples accumulated in the chart's data buffer at run time without resetting the whole chart, an application can set *BufferSize* to -1 and invoke the *GlgUpdate* or *GlgSetupHierarchy* methods to flush the buffer, then set *BufferSize* to a previous or new value.

Y Axes Offset

Specifies an offset used for the layout of multiple Y axes. A negative value defines a pixel offset between the Y axes. A positive value defines an absolute pixel width used by each Y axis. If an axis's width exceeds this positive value, it may overlap the neighboring Y axis.

A negative value defining a pixel offset between the axes may be used to let the chart automatically lay out Y axes. However, it may cause a "jumping" effect if the change of the axis's ranges changes its labels, which in turn changes the width of the axis's box and cause other axes to move. To avoid this effect, an absolute axis width may be specified using a positive value.

Tooltip Mode

Specifies the sample selection priority when processing the chart tooltip or processing a chart selection via an API method. Possible values are:

NONE

Disables chart tooltips

X

Selects the data sample closest to the cursor in the horizontal direction.

XY

Selects the data sample with the minimal distance from the cursor.

Tooltip Format

Specifies a custom format for a chart tooltip. The tooltip format can contain ordinary characters as well as conversion specifications. The conversion specifications contain a source keyword and a conversion format separated by the ':' character, and are surrounded with angle brackets. For example: *"<plot_name:%s>"*.

Depending on the type of the conversion specification source, one of the following matching format types can be used:

double format: some variant of the "%f" format

string format: some variant of the "%s" format

time format: uses the same notation as the *TimeFormat* attribute described on page 126.

The following conversion specifications are supported:

plot_annotation

Replaced with the selected plot's annotation formatted with the supplied string format.

plot_name

Replaced with the selected plot's name formatted with the supplied string format.

plot_string

Replaced with the annotation of the selected plot, or with the plot name if the selected plot has no annotation. The final string is formatted with the supplied string format.

input_x

Replaced with the value corresponding to the cursor's horizontal position and formatted with the supplied double format. For charts with time axes, the value represents a number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC.

sample_x

Replaced with the time stamp or X value of the selected sample formatted with the supplied double format. For time stamps, the value represents a number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC.

input_x_string

input_x_text

Replaced with a string showing the time or X value corresponding to the cursor's horizontal position in the format of the X axis's major tick labels. The final string is then formatted with the supplied string format. The *input_x_string* also removes any line breaks to form a single-line string.

sample_x_string

sample_x_text

Replaced with a string showing the time or X value of the selected sample in the format of the X axis's major tick labels. The final string is then formatted with the supplied string format. The *sample_x_string* also removes any line breaks to form a single-line string.

input_time

Replaced with a string showing the time or X value corresponding to the cursor's horizontal position in the format of the X axis's tooltip. The final string is then formatted with the supplied string format.

sample_time

Replaced with a string showing the time or X value of the selected sample in the format of the X axis's tooltip. The final string is then formatted with the supplied string format.

input_x_time (time charts only)

Replaced with a string showing the time corresponding to the cursor's horizontal position formatted with the supplied time format.

sample_x_time (time charts only)

Replaced with a string showing the time stamp of the selected sample formatted with the supplied time format.

input_time_ms (time charts only)

Replaced with the value showing the number of fractional seconds in the time corresponding to the cursor's horizontal position. It is formed by formatting the number of milliseconds with the supplied double format.

sample_time_ms (time charts only)

Replaced with the value showing the number of fractional seconds in the time stamp of the selected sample. It is formed by formatting the number of milliseconds with the supplied double format.

input_y

Replaced with the Y value corresponding to the cursor's vertical position (in the Low/High range of the selected plot) and formatted with the supplied double format.

sample_y

Replaced with the Y value of the selected sample formatted with the supplied double format.

sample_valid

Replaced with the "yes" or "no" strings depending on the *Valid* flag of the selected sample. The final string is formatted with the supplied string format.

A chart's tooltip is activated by adding a tooltip action to the chart object using the *Object, Add Tooltip* menu option, setting the action's *Tooltip* property to *"$ChartTooltip"* and setting the *ProcessMouse* attribute of the chart's parent viewport to include the *Tooltip* mask. This is the default for all charts in the *Real-Time Charts* palette. The content of the chart's *TooltipFormat* attribute is then used to format the tooltip. If the value of *TooltipFormat* is not set, the following value is used as a default:

<axis_string:%s> axis value= <axis_value_string:%s>

Draw Order

Defines the drawing order of the chart's elements, can have the following values:

Edge,Grid,Plot,Level
Edge,Grid,Level,Plot
Grid,Level,Plot,Edge
Grid,Plot,Level,Edge
Edge,Level,Plot,Grid
Edge,Plot,Grid,Level
Level,Plot,Grid,Edge
Plot,Grid,Level,Edge

The *Edge* refers to the outline of chart's drawing area. For example, using the **Edge,Grid,Plot,Level** setting will draw the outline and the grid first, draw plot lines on top of the grid and then draw level lines on top of the plot lines.

The above *DrawOrder* values are shown in the Graphics Builder. The *GlgChartElemDrawOrder* enum contains corresponding bit masks that can be used to set the value of the attribute via the *GlgSetDResource* method of the GLG API.

Draw Cross-Hair

Enables or disables the cross-hair lines that follow the mouse when the mouse moves over the chart, can have the following values:

NONE
Disables the cross-hair cursor

X
Enables the horizontal cross-hair cursor

Y
Enables the vertical cross-hair cursor

XY
Enables both the horizontal and vertical cross-hair lines.

Edit Grid

A button in the *Properties* dialog for editing attributes of the chart's cross-hair cursor.

Edit SelectionMarker

A button in the *Properties* dialog for editing attributes of the *Marker* object used to annotate the selected data sample. The marker is displayed at the selected data sample when the sample is selected via the API call or by displaying a tooltip. Use the marker's *Visibility* attribute to enable or disable the display of the selection marker.

The chart can be zoomed and scrolled in the X direction by setting the *Span* and *EndValue* attributes of its X axis. Zooming and panning in the Y direction is done by changing the Y ranges of the chart's plots and Y axes.

The *GlgSetZoom* function provides a simplified interface to the integrated zooming and scrolling in the Chart Zoom Mode, which is the default for charts in the Real-Time Charts palette. Refer to the *GlgSetZoom* section on page 89 of the *GLG Programming Reference Manual* for details.

The chart can also be scrolled using integrated scrollbars. The scrollbars are activated by setting the *Pan* attribute of the chart's parent viewport, see page 85.

All charts in the *RealTime Charts* palette have chart tooltips enabled. The chart tooltips display information about the data sample selected by the current cursor position. The type of the information displayed in the tooltips is defined by the chart's *TooltipFormat* attribute. The *TooltipMode* attribute controls the criterion for the data sample selection.

The chart generates messages when its cross-hair cursor is drawn or erased, see the *Chart Message Object* section on page 367 of the *GLG Programming Reference Manual*.

Plot

The *Plot* object is used to render individual lines in a chart. A required number of plot objects is automatically created by the parent chart. A plot object does not have any control points, since its geometry is completely defined by its parent chart. A plot object has the following attributes:

Plot Type
Defines the type of the graphics used to visualize the plot's data, can have the following values:
GLG_LINE_PLOT
GLG_STEP_PLOT
GLG_MARKERS_PLOT
GLG_LINE_AND_MARKERS_PLOT
GLG_STEP_AND_MARKERS_PLOT

Annotation
Specifies a label used to annotate the plot in a chart's legend.

Enabled
Enables or disables the plot. Disabled plots are not shown in a chart, but still keep data in their history buffers and accept new data samples, to be shown when the plot is enabled again.

Y Low
The low range of the plot. If the chart's *CommonRange* is set to YES, the range is shared by all plots in the chart.

Y High
The high range of the plot. If the chart's *CommonRange* is set to YES, the range is shared by all plots in the chart.

Value Entry Point
An entry point for supplying Y values for the plot's samples. The value must be supplied for each data sample pushed into the plot.

Time Entry Point
An entry point for supplying Time or X values for the plot's samples. These values define samples' placement in the horizontal direction. In scrolling charts with *AutoScroll*=YES, pushing a value into the time entry point also scrolls the chart horizontally to display the new data sample.

In charts with the ABSOLUTE TIME SCROLL horizontal axis, the entry point is used to push time stamps for the chart's samples. Supplying a time stamp is optional. If a time stamp is not supplied, the chart automatically generates a time stamp using the current time. A time stamp is supplied in the format of POSIX time (a number of seconds since midnight,

January 1, 1970). Fractional values may be used to supply exact high-precision time stamps.

In charts with the RELATIVE TIME SCROLL horizontal axis, the entry point is used the same way, but the axis labels display the time elapsed since the moment defined by the *TimeOrigin* attribute of the axis.

In charts with the VALUE SCROLL horizontal axis, values pushed into the entry point are used to position data samples based on an arbitrary numerical value other than time.

In charts with the INDEX SCROLL horizontal axis, an application should not push values into the entry point since it is automatically filled with an incrementing sample index: 0, 1, 2 and so on.

In XY Scatter charts with the RANGE horizontal axis, the entry point is used to push X values of XY plots.

Valid Entry Point

An entry point for supplying a sample's VALID attribute (0 or 1). If it is not supplied, the value of 1 is used, making the sample valid by default.

AutoScale Delta

Enables auto-scaling of the plot if set to a non-zero value. If auto-scale is enabled, the plot's range is automatically extended by an integer number of the *AutoScale Delta* intervals to accommodate out-of-range data. If the attribute is set to a negative value, it defines the adjustment delta as a fraction of the plot range. For example, for a value of 0.15, 15% of the plot range will be used as an adjustment interval.

The *Linked Y Axis* attribute of the plot may be used to link the plot with an axis, so that the axis range is adjusted synchronously with the plot when the plot is auto-scaled.

Linked Y Axis

This button is used in the Graphics Builder to associate the plot with a specific Y axis in a chart with multiple Y axes and *CommonRange* set to NO. When the plot range is changed, the range of the associated axis will also change to display the same range, and vice versa. The text box next to the button shows the label string of the associated Y axis.

The *GlgSetLinkedAxis* method of the GLG API may be used at run time to associate a plot with an axis programmatically.

FilterType

Specifies the type of a filter to be used for plots with large number of data points. For example, when a plot shows 50000 data samples on a display that has only 1000 pixels in width, the plot will render 50 points per each pixel in the horizontal direction. In this case, a filter may be used to combine the values of the data samples that fall within the same pixel into a single data sample (or two data-samples for the MIN_MAX filter). Using a filter increases performance and decreases the CPU load by decreasing the number of plot segments and markers that need to be rendered. The following filter types are supported:

NONE

Disables data filtering to draw all data samples.

MIN_MAX

Combines multiple data samples into two data samples that hold the minimum and maximum values of the combined samples.

AVERAGE
Combines multiple data samples into a single data sample by averaging the values of the combined samples.

DISCARD
Plots the first encountered data sample, discarding any other samples that fall onto the same pixel in the horizontal direction.

CUSTOM
Allows the use of custom filters. A custom chart filter can be added to a plot via the *GlgSetChartFilter* GLG API method. Examples of the custom filter code for various programming environments are provided in the *src* subdirectory of the GLG installation. The examples demonstrate the implementation of the above filter types and may be modified to fine-tune filter behavior to suit the application requirements.

FilterPrecision
Specifies the horizontal interval in pixels for combining multiple data samples. The default is 2 pixels, which will cause all data samples in each 2 pixel interval to be combined in one or two data samples depending on the filter type.

FilterMarkers
If set to NO, disables filtering for data samples with markers. It may be used to let data samples with markers through for plots which use markers to annotate special events. If a data sample with a marker is encountered, the data samples for the currently accumulated filter interval are flushed to draw the combined data sample(s), the data sample with a marker is drawn, and then the next filter interval is started.

If *FilterMarkers*=YES, the combined data sample(s) representing the output of the filter for this filter interval will show a marker if any data samples in the interval had a marker.

Include Zero
If set to YES, the plot's range queries always include 0 in the plot's range. For example, if a plot contains data samples in the range [20;100], the range will be reported as [0;100]. If the Y scrollbar is activated, it will scroll the plot from 0 to 100, instead of the [20;100] range. Resetting the chart's Y range via integrated zooming will reset the range to [0;100] as well.

Range Lock
If set to YES, prevents the plot's range from being changed when the chart is zoomed or scrolled in the vertical direction. It may be used for locking plots that display boolean ON/OFF data, so that other plots in a chart can be zoomed and scrolled in Y direction while the locked plot does not change its vertical scale and location. This makes it possible to zoom and scroll other plots vertically and view them in the context of the boolean ON/OFF data, as shown in the GLG Real-Time Strip-Chart demo.

Number of Samples
A read-only attribute containing the number of samples accumulated in the plot. When the chart is reset, the samples are discarded and the number of samples is reset to 0.

Edit Line
A button in the *Properties* dialog to access a *Line Attributes* object that defines the plot's rendering attributes, such as *EdgeColor* and *LineWidth*. These line attributes are inherited by the plot object and are visible as resources of the plot in the resource browser.

Edit Marker
A button in the *Properties* dialog to access a marker object that defines rendering attributes of the plot's markers. The marker's *Visibility* attribute may be used as an entry point to switch visibility of individual markers on or off by supplying 0 or 1 *Visibility* values for

each data sample to annotate selected data samples with markers. If the Visibility value is not supplied, the current value of the attribute will be stored as the marker visibility of the data sample.

Array
A resource of a plot that provides programmatic access to the list of data samples in the plot's history buffer. This attribute is not visible in the Builder, but can be used in a program via the GLG API. The *GlgRTChartMarkers* coding example provides an example of using this resource to toggle marker visibility of the data sample selected with the mouse.

A data sample value pushed into one of the entry points (for example, a value entry point) is buffered in order to wait for data to be supplied for other entry points, such as time or valid entry points. The buffered data sample is flushed and added to the plot when either the *GlgUpdate* method is called, or when data is repeatedly pushed into an entry point which already contains a previously buffered value.

To add a single sample to a plot, push a value into the *ValueEntryPoint* with the *GlgSetDResource* method, push values into the *TimeEntryPoint* and *ValidEntryPoint* if necessary, then invoke the *GlgUpdate* method to update the drawing.

To add multiple samples (for example, when filling the whole chart with data), an application can repeatedly push values into the entry points and invoke the *GlgUpdate* method once at the end.

A plot can be erased without losing the accumulated data points by setting its *Opacity* (the *Opacity* of its *Line Attributes*) to 0. To display the plot again, set its *Opacity* to 1.

Level Line

The *Level Line* object annotates a chart threshold by drawing a horizontal line at a specified height. A required number of Level Line objects is automatically created by the parent chart. A level line does not have any control points, since its geometry is completely defined by its parent chart. A level line object inherits *LineAttributes* such as *EdgeColor*, *LineType*, *LineWidth*, *Opacity* and *AntiAliasing,* and has the following additional attributes:

Level
Specifies the threshold's value.

Y Low
The low range of the level line. If the chart's *CommonRange* is set to YES, the range is shared with the chart's plots.

Y High
The high range of the level line. If the chart's *CommonRange* is set to YES, the range is shared with the chart's plots.

Enabled
Enables or disables the display of the level line.

Range Lock
If set to YES, prevents the level line's range from being changed when the chart is zoomed or scrolled. It may be used for locking the vertical position of level lines corresponding to the locked plots that display boolean data. If a level line is locked, its position does not change when the chart is scrolled or zoomed in the Y direction.

Linked Y Axis

This button is used in the Graphics Builder to associate the level with a specified Y axis in a chart with multiple Y axes and *CommonRange* set to NO. The range of the level line will be automatically changed to correspond to the range of the associated Y axis when the range of the axis changes, and vice versa. The text box next to the button shows the label string of the associated Y axis.

The *GlgSetLinkedAxis* method of the GLG API may be used at run time to associate a level line with a Y axis programmatically.

Line attributes of a level line can be reused the same way as described in the *Line Attributes* section on page 144.

Axis

The *Axis* object is used by a chart to draw X and Y axes. It may also be used as a stand-alone axis or ruler object. An axis object is defined by two points. For an axis that is a part of a *Chart* object, the axis's points are controlled by the chart and positioned in the diagonal corners of the chart's data area. The *Visibility* attribute of an integrated chart axis can be used to turn individual axes on or off.

An axis object supports integrated tooltips that can be used to display an axis value corresponding to the current mouse position. The content of the tooltip string is controlled by the axis's *TooltipFormat* attribute.

Axis attributes depend on the type of the axis: some attributes are common for all axis types, and other attributes are specific for a chosen axis type. In the Builder, the *Property* dialog displays only the attributes applicable to the selected axis type. The following lists all attributes of the axis object:

Axis Type

Defines the type of the axis. The type is a combination of several constants that define the axis behavior:

RULER

Controls scaling of the axis's tick intervals. A ruler axis positions its minor and major ticks at pixel coordinates. When the length of the axis is increased or decreased, the axis keeps the pixel length of tick intervals constant and changes its displayed span to show a bigger or smaller number of ticks. The *RulerStart* and *RulerScale* attributes control the origin and the scale of the ruler.

If the RULER modifier is not present and the axis's length is increased or decreased, the axis preserves the visible span and proportionally changes the pixel length of tick intervals to maintain the number of displayed ticks constant.

RANGE

Defines a value axis controlled by the Low and High attributes.

SCROLL

Defines a scrollable axis with a visible span controlled by the *EndValue* and *Span* attributes.

TIME, VALUE or INDEX modifiers

Specify an interpretation and formatting of the values displayed in the scrolling axis's labels. Depending on the used modifier, the values are interpreted and displayed as either a time stamp, a double value or an integer sample index.

LOCAL, UTC or RELATIVE

Modify behavior of a scrolling time axis to display either an absolute date and time (LOCAL or UTC), or a time interval elapsed since the time specified by the *TimeOrigin* attribute (RELATIVE).

CENTER modifier

Controls positioning of the axis's major and minor ticks. If present, ticks are positioned in the middle of the tick intervals as opposed to being positioned at the end of each tick interval.

The actual axis type is a combination of the constants listed above. The following demonstrates a few common examples:

- An ABSOLUTE TIME SCROLL axis type defines a scrolling time axis that displays a span of time defined by the *Span* attribute. The end position of the displayed time span is controlled by the *EndValue* attribute.

- An ABSOLUTE TIME RULER SCROLL axis type defines a scrolling time axis that displays ticks and labels at fixed pixel intervals regardless of the axis's length. The *RulerScale* attribute controls the seconds-to-pixels mapping. For example, if *RulerScale*=1, each pixel will correspond to one second. If *RulerScale*=5, the length of one second interval will be 5 pixels.

- A RANGE axis type defines a value axis with a range supplied by the *Low* and *High* attributes; this axis type is used for the Y axes of a chart.

- A RULER axis type defines a ruler. The *RulerScale* attribute defines the ruler's unit-to-pixel mapping. If *RulerScale* is set to the DPI (Dots Per Inch) value of the monitor and the axis's *MajorInterval*=1, the axis will display major ticks at 1-inch intervals.

Axis Position

Specifies the axis placement relatively to a rectangle defined by the axis's two control points and may have the following values:

HTOP UP

HTOP_DOWN

A horizontal axis is positioned along the top edge of the rectangle defined by the control points, with ticks drawn in the direction specified by the UP or DOWN keyword.

HCENTER UP

HCENTER DOWN

A horizontal axis is positioned in the center of the rectangle defined by the control points, with ticks drawn in the direction specified by the UP or DOWN keyword.

HBOTTOM UP

HBOTTOM DOWN

A horizontal axis is positioned along the bottom edge of the rectangle defined by the control points, with ticks drawn in the direction specified by the UP or DOWN keyword.

VLEFT LEFT

VLEFT RIGHT

A vertical axis is positioned along the left edge of the rectangle defined by the control points, with ticks drawn in the direction specified by the trailing LEFT or RIGHT keyword.

VCENTER LEFT

VCENTER RIGHT

A vertical axis is positioned in the center of the rectangle defined by the control points, with ticks drawn in the direction specified by the trailing LEFT or RIGHT keyword.

VRIGHT LEFT

VRIGHT RIGHT

A vertical axis is positioned along the right edge of the rectangle defined by the control points, with ticks drawn in the direction specified by the trailing LEFT or RIGHT keyword.

When a chart's X or Y axis is edited in the Builder, only the options applicable to the selected axis type are included in the menus for editing the axis's position.

Inversed

If set to YES, inverts the axis direction. The default direction is left to right for a horizontal axis, and bottom to top for a vertical axis.

Draw Outline

Controls the drawing of auxiliary decorations, can have the following values:

NO_OUTLINE

No additional decorations are drawn.

AXIS_LINE

A line is drawn along the axis.

BOX_MINOR

Draws a box around minor ticks.

BOX_MAJOR

Draws a box around major ticks.

BOX_ALL

Same as BOX_MAJOR and BOX_MINOR used together.

LINE_MINOR

Draws a line along the outer ends of minor ticks.

LINE_MAJOR

Draws a line along the outer ends of major ticks.

LINE_ALL

Same as LINE_MAJOR and LINE_MINOR done together.

FillType and *FillColor* of the *Ticks and Outline* attributes may be used to draw filled outline boxes.

End Value (scrollable axes only)

Defines the value that corresponds to the end of a scrolling axis. An axis is scrolled by dynamically changing the value of this attribute.

Span (scrollable axes only)

Defines the extent (in axis units) from the beginning of the axis to its end. For example, setting *Span*=60 for a chart's time axis will display 60 seconds worth of data, ending with the time specified by the *EndValue* attribute.

Low (range axes only)
Specifies the value corresponding to the axis start.

High (range axes only)
Specifies the value corresponding to the axis end.

Ruler Start (ruler axes only)
Specifies the value corresponding to the axis start.

Ruler Scale (ruler axes only)
Defines a scale factor for unit-to-pixel mapping. For example, if *RulerScale*=10, one unit of the ruler will be mapped to 10 pixel. To display an inch ruler, set *RulerScale* to the value of the monitor's DPI (Dots Per Inch). To display a metric centimeter ruler, set *RulerScale* to the value of the monitor's DPI divided by 2.54.

Time Format (time axis only)
Specifies a format for displaying time labels. The format string follows the convention of the POSIX *strftime* function and can contain ordinary characters as well as conversion specifications. The conversion specifications are two character sequences that start with the '%' character and are replaced by formatted time and date strings as described below.

The conversion specifications listed below are universally supported across Windows, Linux/Unix, Java and C#/.NET environments.

Escape sequences may be used to define native platform-specific formats for Windows and Unix platforms, as well as Java and C#/.NET environment, as described in the *Scalar Formatting (Format D)* section on page 164.

Some conversion specifications may not be supported on Unix platforms other than Linux, such as Solaris, HPUX, AIX, etc. Refer to the manual page of the native *strftime* function for information on the supported conversion specifications for these Unix platforms.

%a
The abbreviated weekday name according to the current locale.

%A
The full weekday name according to the current locale.

%b
The abbreviated month name according to the current locale.

%B
The full month name according to the current locale.

%c
The preferred date and time representation for the current locale.

%d
The day of the month as a decimal number (range 01 to 31).

%D
Equivalent to %m/%d/%y.

%e
Like %d, the day of the month as a decimal number, but a leading zero is replaced by a space.

%F
Equivalent to %Y-%m-%d.

%h
Equivalent to %b.

%H
The hour as a decimal number using a 24-hour clock (range 00 to 23)

%I
The hour as a decimal number using a 12-hour clock (range 01 to 12)

%j
The day of the year as a decimal number (range 001 to 366).

%k
The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. (See also %H.)

%l
The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. (See also %I.)

%m
The month as a decimal number (range 01 to 12).

%M
The minute as a decimal number (range 00 to 59).

%n
A newline character.

%p
Either “AM” or “PM” according to the given time value, or the corresponding strings for the current locale. Noon is treated as “PM” and midnight as “AM”.

%r
The time in a.m. or p.m. notation. In the POSIX locale this is equivalent to %I:%M:%S %p.

%R
The time in 24-hour notation (%H:%M). For a version including the seconds, see %T below.

%S
The second as a decimal number (range 00 to 60). (The range is up to 60 to allow for occasional leap seconds.)

%t
A tab character.

%T
The time in 24-hour notation (%H:%M:%S).

%x
The preferred date representation for the current locale without the time.

%X
The preferred time representation for the current locale without the date.

%y
The year as a decimal number without a century (range 00 to 99).

%Y
The year as a decimal number including the century.

%z
The time-zone as hour offset from GMT.

%Z
The time zone or name or abbreviation.

%%
A literal '%' character.

The following conversion specifications are supported only in some environments:

%C
The century number (year/100) as a 2-digit integer. (Not supported on Windows.)

%P
Like %p but in lowercase: "am" or "pm" or a corresponding string for the current locale (except on Windows, where it is the same as %p).

%s
The number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC (except on Windows where it is the same as %c).

%U
The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. (Not supported in Java and C#/.NET.)

%w
The day of the week as a decimal, range 0 to 6, Sunday being 0. See also %u. (Not supported in Java.)

%W
The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01. (Not supported in Java and C#/.NET.)

For example, the "Time=%X%nDate=%x" may display the following label in the US locale:

Time: 12:49:13 PM
Date: 02/06/2013

MilliSec Format (time axes only)
Specifies a double-precision C-style format for an optional display of fractional seconds in the form of milliseconds at the end of the axis labels. It may be set to an empty string to suppress milliseconds display.

For scrolling axes with relative time, the default setting is ".%03.0f", which will display 270.5 milliseconds as ".270".

For scrolling axes with absolute time, *MilliSecFormat* is set to an empty string by default and is not displayed in the Builder's *Property* dialog. The attribute can still be accessed via the *Resource Browser* using the "MilliSecFormat" resource name.

Label Format (non-time axes only)
Specifies a double-precision C-style format for axis labels (non-time axes only). For example, the "x=%.0f" format will display 12.5 as "x=12".

Major Interval
Specifies an interval between major ticks. A positive value may be used to specify an interval in axis units. A negative value may be used to specify a fixed number of major ticks instead of an interval, which may be convenient for automatic tick positioning in cases when the axis range may change. A zero value may be used to disable major ticks and labels.

For example, if *MajorInterval*=3600 for a time axis, the major ticks and labels will be

placed at 1-hour intervals. If the axis's *Span* is increased to display 7 days worth of data, the axis will display an excessive number of major ticks and labels. If *MajorInterval*=-5, the axis will always display 5 major ticks and labels regardless of the axis's *Span*.

Minor Interval

Specifies an interval between minor ticks. A positive value may be used to specify an interval in axis units. A negative value may be used to specify the number of minor ticks per one major tick interval. A zero value may be used to disable minor ticks.

AxisLabel String

Specifies an axis label that may be used to identify the axis, for example in a chart with multiple Y axes.

Edit Ticks & Outline

A button in the *Properties* dialog to access the *LineAttributes* object that defines rendering attributes of the ticks and the outline.

Edit Ticks Labels

A button in the *Properties* dialog to access a text object that defines attributes of the major tick labels.

Low Offset (non-time axes only)

High Offset (non-time axes only)

Decrease the visible part of the axis while keeping its *Low/High* ranges. This may be used to display only a part of the whole axis, as it is done for the axis of the boolean plot in the GLG Real-Time Strip-Chart demo. The axis for the boolean plot shows only a small part of the axis corresponding to the [0;1] interval.

The offsets are defined in relative units in the range from 0 to 1. *LowOffset* specifies an offset at the low end of the axis, and *HighOffset* at the high end. The sum of the low and high offsets should not exceed 1. For example, for a range axis with *Low*=0, *High*=100, *LowOffset*=0.1 and *HighOffset* = 0.2, only a part of the axis from 10 to 80 will be displayed. 10% at the low end of the axis and 20% at the high end will be truncated due to the settings of the low and high offsets.

The following lists the equations for determining the location of the visible ends of the axis when High is greater than Low:

```
visible_low = Low + LowOffset * ( High - Low );
visible_high = High - HighOffset * ( High - Low );
visible_interval = ( High - Low ) * ( 1. - LowOffset - HighOffset );
```

The following equations can be used to calculate the low and high offsets based the required values for the visible ends of the axis:

```
LowOffset = ( visible_low - Low ) / ( High - Low )
HighOffset = ( High - visible_high ) / ( High - Low )
```

Rounded Placement

If set to YES, the axis places major ticks and labels at positions corresponding to values that can be evenly divided by the major tick interval. If set to NO, the first major tick will be placed exactly at the start of the axis.

For example, for a range axis with *RoundedPlacement*=YES, *MajorInterval*=10 and *Low*=8, the major ticks and labels will be positioned at the following values: 10, 20, 30 and so on. If *RoundedPlacement* is set to NO with the rest of the attributes unchanged, the major ticks and labels will be positioned at: 8, 18, 28 and so on.

For a time axis with *RoundedPlacement*=YES and *MajorInterval*=3600, the major ticks and labels will be placed at exact hourly positions regardless of the value of the axis's *EndValue* attribute.

The rounded placement is most useful when the major tick interval is explicitly defined by setting MajorInterval to a positive number. When MajorInterval is negative, the result of the rounded placement may look confusing, since it defines a number of major ticks and labels displayed in the visible span of the chart. In this case the actual major interval may be non-integer, and the result of the rounded placement may look confusing.

Fix Leap Years (time axes only)
If set to YES, activates a leap year adjustment that improves the year display accuracy for axes that span a large multi-year period.

Major Tick Size
Specifies the length of the major tick in pixels.

Minor Tick Size
Specifies the length of the minor tick in pixels.

TickLabel Offset
Specifies a pixel offset between a major tick and its label.

Label Extent Relative (advanced)
Specifies an extent of a major tick label in the direction parallel to the axis. It is used for scaling tick labels with *TextType* set to SCALED. The extent is relative to the major tick interval: the value of 0.8 would allow tick labels to use 80% of the major tick interval. If a label does not fit, it is scaled down.

Label Extent Absolute (advanced)
Specifies a pixel extent of a major tick label in the direction perpendicular to the axis. It is used for scaling tick labels with *TextType* set to SCALED. If a label does not fit, it is scaled down.

Major Offset
Specifies an additional offset for positioning major ticks and labels. For example, for a range axis with *RoundedPlacement*=YES, *Low*=8, *MajorInterval*=10, setting *MajorOffset*=5 will position major ticks and labels at 15, 25, 35 and so on instead of 10, 20, 30 and so on.

If the value of *MajorOffset* exceeds major tick interval, a remainder of an integer division of *MajorOffset* by *MajorInterval* is used as the offset value.

Time Origin (relative time axis only)
Specifies the start time to be subtracted from the sample's time stamps. For example, if a relative time chart is used to display data of an experiment, it may be desirable to display the start time of the experiment as 00:00:00 and the time 12 minutes and 5 seconds after the start of the experiment as 00:12:05. There are two ways to accomplish this. An application can set *TimeOrigin* to 0 and supply relative time stamps as time intervals expired since the start of the experiment. Alternatively, it may set *TimeOrigin* to the time of the experiment's start and supply absolute time stamps: in this case, the axis will subtract the start time from the time stamps before displaying them in the major tick labels. If the X axis of a chart is a relative time axis, the chart will also subtract the start time from the time stamps when plotting the data points, while keeping the original absolute time stamps in the data buffer.

Tooltip Format

Specifies a custom format for an axis tooltip. The tooltip format can contain ordinary characters as well as conversion specifications. The conversion specifications contain a source keyword and a conversion format separated by the ':' character, and are surrounded with angle brackets.

Depending on the type of the conversion specification source, one of the following matching format types can be used:

double format: some variant of the "%f" format

string format: some variant of the "%s" format

time format: uses the same notation as the *TimeFormat* attribute described on page 126.

The following conversion specifications are supported:

axis_label
Replaced with the axis label formatted with the supplied string format.

axis_name
Replaced with the axis name formatted with the supplied string format.

axis_string
Replaced with the axis label, or with the axis name if the axis has no label. The final string is formatted with the supplied string format.

axis_value
Replaced with the value corresponding to the cursor position and formatted with the supplied double format. For time axes, the value represents a number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC.

axis_value_string

axis_value_text
Replaced with a string showing the axis time (or the value corresponding to the cursor position for range axes) in the format of the axis's major tick labels. The final string is then formatted with the supplied string format. The *axis_value_string* also removes any line breaks to form a single-line string.

axis_time (time axes only)
Replaced with a string showing the time corresponding to the cursor position formatted with the supplied time format.

axis_time_ms (time axes only)
Replaced with the value showing the number of fractional seconds in the time value that corresponds to the cursor position. It is formed by formatting the number of milliseconds with the supplied double format.

For integrated axis objects of a chart, the tooltip is activated by the chart's tooltip settings.
For a stand-alone axis, an axis's tooltip is activated by adding a tooltip action to the axis object using the *Object, Add Tooltip* menu option, setting the action's *Tooltip* property to *"$AxisTooltip"* and setting the *ProcessMouse* attribute of the axis's parent viewport to include the *Tooltip* mask. . The content of the axis's *TooltipFormat* attribute is then used to format the tooltip. If the value of *TooltipFormat* is not set, the following value is used as a default:

<axis_string:%s> axis value= <axis_value_string:%s>

AxisLabel Offset
Specifies additional X and Y offsets for the axis label. The Z offset is ignored.

AxisLabel Position
Specifies the anchoring of the axis label's control point relatively to the axis's bounding box.

AxisLabel Anchor
Specifies the anchoring of the axis label's string relatively to the label's control point.

Edit Axis Label
A button in the *Properties* dialog to access a text object that defines attributes of the axis label.

Legend

The *Legend* object provides information that helps identify data displayed in a chart. For each plot, the legend displays a label and a line that shows an example of the plot appearance. A label displays a string specified by the plot's *Annotation* attribute; if an annotation is not supplied, the plot name is used.

While the legend is tightly coupled with its chart, it is implemented as a separate object to allow developers to control its location and layout. A legend is attached to a chart by selecting it, marking it with the *Arrange, Legend, Mark Legend* menu option, then selecting a chart and using the *Arrange, Legend, Set Chart Legend* menu option. The *Arrange, Legend, Reset Chart Legend* menu option can be used to disconnect the legend from the selected chart.

The legend has two control points that define its bounding box. The legend content may extend beyond the box if it does not provide enough space to lay out the legend's entries. The legend may be placed in a separate viewport if it is desirable to clip the legend's content when it becomes too long. When a legend is placed in a separate viewport, that viewport and not the legend itself must be used for marking the legend with the *Arrange, Legend, Mark Legend* menu option.

The legend has the following attributes that control its appearance and layout:

Layout Type
Specifies the type of layout used for positioning legend entries, can be one of the following:

HORIZONTAL
Legend entries are laid out left to right along a single horizontal line.

HORIZONTAL WRAPPED
Legend entries are laid out horizontally. If the legend extends beyond the box defined by its control points, it is wrapped to start a new horizontal row below the previous one.

VERTICAL
Legend entries are laid out top to bottom along a single vertical line.

VERTICAL WRAPPED
Legend entries are laid out vertically. If the legend extends beyond the box defined by its control points, it is wrapped to start a new vertical line to the right of the previous one.

Auto Layout (wrapped legends only)
Controls row wrapping behavior of the legend as described below. The term "row" is used for both horizontal and vertical lines of legend entries regardless of their direction.

By default, *AutoLayout* is set to YES and the legend wraps a row when one of the following conditions is satisfied:

- *MinRowSize*=0 and a row extends beyond the box defined by the legend's points
- *MinRowSize* is greater than zero, a row extends beyond the legend's box and the number of entries in the row is greater or equal to *MinRowSize*
- *MaxRowSize* is greater than zero and the number of entries in the row is greater or equal to *MaxRowSize.*

If *AutoLayout* is set to NO and *MaxRowSize* is not zero, wrapped legends will ignore the minimum row size and wrap rows only when the row size reaches the maximum row size, even if legend rows extend beyond the box defined by the legend's points.

If the legend's labels use scalable text, the legend may also use a smaller font for rendering labels if the legend's content does not fit the legend's box.

Anchor

Specifies the anchoring type of the legend's content relatively to the box defined by the legend's points.

Row Anchor

Specifies the anchoring type of individual rows of a multi-row legends with rows that have different pixel length.

Min Row Size

Specifies the minimum number of elements in a row for wrapped legends. If the number of elements in a row is less than the minimum, a new row will not be started even if the row does not fit into the legend's box.

Max Row Size

Specifies the maximum number of elements in a row for wrapped legends. If the number of elements in a row is more than the maximum, a new row will be started even if the row fit into the legend's box and can accommodate more entries.

Edit Labels

A button in the *Properties* dialog to access a text object that defines attributes of the legend's labels.

Edit Background

A button in the *Properties* dialog to edit attributes of a background box drawn around the legend's content. To disable the display of the background, set its *FillType* to *NONE* or set its *Opacity* to 0.

Line Length

Specifies a length in pixels of the line element of a legend entry.

Min Line Width

Specifies a minimum line width of the line element of a legend entry.

XOffset

YOffset

Specifies horizontal and vertical pixel offsets between the edge of the legend's background box and the legend content.

XSpacing

YSpacing

Specifies horizontal and vertical pixel spacing between the elements of the legend.

Label XOffset

Specifies a horizontal pixel offset between the line and label elements of a legend entry.

Label Max Width
Specifies the maximum pixel width for scalable legend labels with *TextType*=SCALED. If a label does not fit, it is scaled down.

Label Max Height
Specifies the maximum pixel height for scalable legend labels with *TextType*=SCALED. If a label does not fit, it is scaled down.

Non-Graphical Objects

While GLG non-graphical objects are not visible in a GLG drawing, they can control the position and a wide variety of visible features of the objects that do appear. *These objects are for advanced usage and may be safely ignored until they are really needed.*

Data

The data object is used to specify a data value. Most attributes of GLG objects use data objects to keep an attribute's value.The data objects are also used to attach *Custom Data Properties* to other objects.

The data object has the following attributes: the data type, the value, the transformed value and an optional tag. Of course, it also has a name, the *Global* and *HasResources* flags, and may have transformations attached to it.

A tag may be attached to the data object to mark it as a global resource to define database connectivity for the data value. Refer to the *Tag-Based Data Access and Database Connectivity* chapter on page 60 for details of using tag objects for data access.

If a transformation is attached to a data object, it will transform it causing the transformed value to differ from the original data value. Data objects are not affected by drawing and other transformations attached to their parents.

The attributes of the data object are as follows:

Type
Specifies the type of data stored. Possible data types are G, D, or S, for "geometrical", "double-precision," and "string", respectively. A G value contains three double-precision values. It usually represents a point in a Euclidean space, but it can be used for any data stored in triples (RGB color values, for example). A D value is a single number (also called a "scalar"), rendered in double-precision, and an S value is simply a standard character string.

Value
Where the actual data value is stored. Note that *Value* is omitted from a resource name when the value of the attribute represented by the data object is accessed. For example, *"my_object/Angle"* resource name is used to access an object's Angle parameter, and not *"my_object/Angle/Value"*. NULL may be used as a resource name to access the value of a data object using its object ID, as shown in the following example:

```
GlgObject data_obj;
data_obj = GlgGetResourceObject( my_object, "Angle" );
GlgSetDResource( data_obj, NULL, 90. );
```

XfValue

A read-only data value as it was transformed by the transformation attached to the data object. If no transformations are attached to the object, the transformed value is the same as *Value*. The transformed value is valid only after the drawing hierarchy has been setup. For G data representing points, the transformed value is in world coordinates. The following example queries the transformed value of an object's attribute:

```
double xf_value;
GlgGetDResource( my_object, "Angle/XfValue", &xf_value );
```

TagObject

An optional tag object that may be attached to the data object to mark it as a global resource or to define database connectivity for its data value. If a tag object is present, the data object inherits all tag attributes, such as *TagName*, *TagSource* and *TagEnabled*.

UTF8Encoding (data objects of S type only)

A boolean flag that defines the encoding used for **storing** the string. If it is set to YES, the string value will be stored using the UTF-8 encoding, otherwise it will be stored using the default encoding defined by the system locale.

For the text objects with the UTF-8 encoded strings to be **rendered** properly regardless of the system locale setting, the text object should use a UTF-8 font with *MultiByteFlag* set to UTF8. Refer to the *Font* section on page 148 for more information. If the UTF8 setting of the text string and the font used to render it do not match, the appropriate encoding conversion will be automatically performed. The characters that are not present in the font will be replaced with the default character.

In the **Java and C#/.NET environments**, the *UTF8Encoding* flag is used only for proper string decoding when the drawing is loaded. Once loaded, the string is stored in memory in Java or C# internal representation, which is using the UTF-16 version of UNICODE. This ensures proper rendering of the string regardless of the system locale, and the *MultiByteFlag* of the font is ignored.

When the *UTF8Encoding* flag is changed in the Builder, the string's encoding is automatically converted from UTF8 to the system encoding or vice versa, depending on the flag's state. If some characters in an UTF8-encoded string can not be represented in the system locale, the conversion is non-reversible and *UTF8Encoding* flag is restored to its original state. Shift-click on the UTF8 toggle to proceed with a non-reversible conversion that results in a loss of information. If the string contains invalid multi-byte characters for the system locale, the conversion fails and the flag is restored to its original value as well. *Ctrl-click* on the UTF8 toggle may be used to change the flag without converting the string; this may result in an invalid character string and should be used only to fix UTF8-encoded strings in old drawings which did not have the *UTF8Encoding* flag.

When setting values of string resources at run time, it is an application's responsibility to set the *UTF8Encoding* flag to a state matching the encoding of the string passed to the *GlgSetSResource* method. Setting the value of the flag at run time using the *GlgSetDResource* method does not re-encode the string.

Note, that unlike most GLG objects attributes, the *Value* of the data object is not stored as an object. This avoids the infinite regression that might exist if an object's attribute was represented as a data object, whose data value was represented as another data object, whose data value was represented by still another data object, and so on. The same is true for the *Name* and the flags (*Global*, *HasResources*, etc.).

Data objects may be used programmatically for attaching custom properties to other objects.When custom properties are attached to objects in the GLG Builder (*Add Custom Property* button of the *Object* Menu), the data objects are used to keep the properties' values.

Attribute

The attribute object is a subclass of the data object used to specify an attribute value. The attribute objects are used to keep values of attributes, control points and values of list transformations. Polygons store their control points as an array of attribute objects.

The attribute object differs from the data object by its transformational behavior. Unlike the data object, it is transformed not only by the transformation attached to the attribute itself, but also by any related transformations attached to its parents. For example, consider a polygon's point with transformations attached to both the point and the polygon. The point is an attribute object and is transformed by both transformations, as well as by the drawing transformation of the viewport in which the polygon is contained.

To access attribute objects via resources, use default attribute names instead of named object attributes. For example, *"my_polygon/LineWidth"* may be used to access the *LineWidth* attribute (vs. *"my_polygon/my_color"*). For polygon points, the *GlgGetElement* function can also be used to access point's attributes.

The attribute object's properties are similar to the properties of the data object:

Type
: Specifies the type of data stored. Possible data types are G, D, or S, for "geometrical", "double-precision," and "string", respectively. A G value contains three double-precision values. It usually represents a point in a Euclidean space, but it can be used for any data stored in triples (RGB color values, for example). A D value is a single number (also called a "scalar"), rendered in double-precision, and an S value is simply a standard character string.

Role
: Determines the type of transformations that will affect the object. This is a creation-time attribute whose possible values include GLG_GEOM_XR (geometrical) and GLG_COLOR_XR (color). The GLG_GDATA_XR, GLG_DDATA_XR and GLG_SDATA_XR values may be used with G, D and S attributes, respectively, for attributes other then points and colors.

Value
: Where the actual attribute value is stored. Note that *Value* is omitted from a resource name when the value of the attribute is accessed. For example, *"MyPolygon/LineWidth"* resource name is used to access the line width of a polygon, and not *"MyPolygon/LineWidth/Value"*. NULL may be used as a resource name to access the value of a data object from its object ID, as shown in the following example:

```
GlgObject attr_obj;
attr_obj = GlgGetResourceObject( polygon, "LineWidth" );
GlgSetDResource( attr_obj, NULL, 2. );
```

XfValue
A read-only attribute value as it was transformed by the transformations attached to the attribute. The transformed value is valid only after the drawing hierarchy has been setup. If no transformations are attached to the object, the transformed value is the same as *Value* (except for G attributes representing points). For G attributes representing points, the transformed value represents the screen coordinates of the point and includes the effect of the all transformations attached to the graphical object containing the point and all its parents, as well as drawing transformation of the viewport. The following example queries the transformed value of an object's attribute:

```
double xf_value;
GlgGetDResource( polygon, "LineWidth/XfValue", &xf_value );
```

TagObject
An optional data tag object that may be attached to the attribute object to mark it as a global resource or to define database connectivity for its data value. If a tag is attached, the attribute object inherits all tags attributes, such as *TagName*, *TagSource* and *TagEnabled*.

ExportTag
An optional export tag object that may be attached to the attribute object to mark it as an exported public property. It is used by the OEM version of the Builder; refer to the *OEM Version of the Graphics Builder* chapter on page 282 for details.

Data
The base data object used for storing the attribute's value.

Tag

The tag object may be attached to any attribute of an object to mark it as a global resource or specify the database field to use for updating the value of the attribute.

The tag object has three attributes:

TagType
Defines the type of a tag. The default DATA tag type is used for data connectivity. The other tag types, EXPORT and EXPORT_DYN, are used by the OEM version of the Graphics Builder to define public properties of components for the GLG HMI configurator and properties of custom dynamics, respectively. The EXPORT_DYN tag type is also used to define public properties of action commands and action data sets. Refer to the *OEM Version of the Graphics Builder* chapter on page 282 for details.

TagName
A string used to identify the tag during browsing. Having a separate *TagName* attribute provides a persistent tag identification regardless of the changing *TagSource* attribute.

TagSource
Defines the database field to use as a datasource for the data object the tag is attached to. A typical application queries a list of *TagSources* defined in the drawing on application startup and uses it to subscribe for data updates from a process database. When data changes, the application sets the new data values by invoking the *GlgSetTag* method, passing the *TagSource* and the new data value for each tag as shown in the source code of the Tag Example. The TagSource attribute is used only by the DATA tags.

TagAccessType

Specifies an access type of the tag, may have the following values:

INPUT

An input tag that may be updated with incoming data. This is the default.

OUTPUT

An output tag used to send data back to the process controller. The output tag is skipped and is not updated when the *SetTag* methods are used. Output tags may be used in an application to keep IDs of tags to be updated in the process database when user enters a new tag value.

INIT

Same as INPUT, used to indicate that an application needs to set the tag just once when the drawing is initially displayed.

TagEnabled

This D attribute may be set to FALSE at runtime to temporarily disable updates of the tag with the *SetTag* methods. The attribute may be used by an application to disable updates of a text input object from an attached tag while the user enters a new value. Disabling a tag does not affect other enabled tags with the same *TagSource*: they will still be updated with new values when the *SetTag* methods are invoked.

TagComment

A string used to hold user-defined information related to the tag.

The *Data Tag* dialog in the Builder also shows the *InLow* and *InHigh* attributes. These attributes do not belong to the tag object itself, but to the attribute object the tag is attached to. If the tag is attached to a D attribute with a range transformation, the *InLow* and *InHigh* fields allow the user to edit the input range of the range transformation right in the tag editing dialog. These fields are disabled otherwise.

The *Browse* button of the *Data Tag* dialog allows the user to browse custom data sources and select tag sources from a list of available choices. A custom data DLL may be provided to connect to the application-specific data sources, such as a process database or PLC controller.

Refer to the *Tag-Based Data Access and Database Connectivity* chapter on page 60 for details of using tag objects for data access.

Using Output Tags and Disabled Tags in a Program

The *SetTag* and *GetTag* methods of the GLG API skip tags that have *TagEnabled=FALSE* , generating an error if no enabled tags with the requested *TagSource* were found in the drawing. The *SetTag* method also generates an error if the only found tag with the requested *TagSource* is an OUTPUT tag.

When the *QueryTags* and *GetTagObject* methods are used in a single tag or unique tags modes, they return the best available tag(s): enabled INPUT and INIT tags have priority over the disabled and OUTPUT tags.

History

The history object is used to control the scrolling behavior of numbered resources. This is most useful for controlling series behavior in graphs, but is general enough to be useful in a variety of situations.

The history object uses an input resource name mask, a **scroll type** and an **entry point**. The scroll type determines the precise behavior of the object. Using the WRAPPED scroll type, each time the entry point attribute is changed, its value is written to the next resource that matches the input mask. The mask uses a percent symbol (%) as a wildcard character. Suppose the input mask is *GraphBar%/Height*. The first time the entry point is changed, the change is written to the resource *GraphBar0/Height*. The second time, the change is written to *GraphBar1/Height*, and the third time to *GraphBar2/Height*. This continues until there are no more matches for the mask, at which point it starts over again with *GraphBar0/Height*.

Setting the scroll type to SCROLLED creates similar behavior, but all changes are initially made to the first object in the series. However, each time the first object is changed, the second object takes the old value of the first, and the third takes the old value of the second and so on. The last data value in the series is discarded.

A history may be attached to an object in the Builder by using the *Add History* button in the *Object* Menu. The history object has four attributes:

ScrollType
: Controls how changes are made to the resources that match the input mask. Using the WRAPPED type, changes in the entry point are made to each of the objects in the series in turn. Using the SCROLLED type, changes are only made to the first object in the series, but with each change in the entry point, the second object takes the old value of the first object, the third takes the old value of the second, and so on. The value of the last object in the series is discarded.

VarName
: The input resource name mask. Use a percent symbol (%) for the variable position. For example, *GraphBar%/Height* will become *GraphBar0/Height* and *GraphBar1/Height* and so on in turn. All resource names are relative to the position of the history object itself. That is, if a history is attached to a series object, the resource name mask need not contain the name of the series itself. This attribute is not an object but a string (*char**). If there is no percent symbol in the VarName string, it is added to the end.

EntryPoint
: This is the entry point for the object. Each time this attribute is changed, its changes are propagated to the list of resources that match the *VarName* attribute.

Inversed
: Determines whether the history object works in the order defined by the resource names (DIRECT) or in reverse order (INVERSED).

RollBack
: Defines the number of iterations to “roll back” when the history gets completely filled with data. This attribute is used in conjunction with the *WRAPPED* scrolling type for implementing scrolling behavior which scrolls the graph only once every **n** iterations, as defined by the value of the attribute.

When the *RollBack* attribute is used in a graph, the *RollBack* attributes of the *DataGroup* object and the *XLabelGroup* object must be set to proper values to ensure their synchronous scrolling. For example, consider a graph with the *WRAPPED* scroll type, 200 data samples, 10 X axis major ticks with labels, and 20 minor ticks per one major tick interval. Setting *DataGroup/Rollback=40* and *XLabelGroup/RollBack=2* will "roll" the graph back by 2 major tick and label intervals (which corresponds to 40 data samples) when the graph gets completely filled with data.

The use of the *RollBack* limits the CPU-intensive scrolling operation to be performed only once on every 40th data update, compared with every data update in the regular scrolling graph with the *SCROLLED* scrolling type.

A special case of the rollback may be used to implement the graph which switches from the WRAPPED behavior to the SCROLLED behavior when the graph gets completely filled with data the first time. For example, for a graph with the WRAPPED scroll type, 200 data samples, 10 X axis major ticks and labels, and 20 minor ticks per one major tick interval, the following settings may be used: *DataGroup*/*Rollback=1* and *XLabelGroup*/*Rollback=0.05* (which corresponds to one minor tick - 1/20).

Alias

An alias object may be used to define logical names for arbitrary resource hierarchies. For example, it may define a logical "ValueHighlight" name for accessing the "Group1/Object3/FillColor" resource hierarchy. The application can then access the resource using the alias instead of a complete path name. The alias can also be used to create convenient shortcuts for long resource paths.

Aliases may be added in the Builder using the *Add Alias* button in the *Object* Menu. The alias object has the following resources:

Alias
Specifies a logical name to be assigned to the resource hierarchy pointed to by the alias.

Path
Resource path to the aliased resource.

Rendering

The rendering object is used to keep an extended set of optional rendering attributes. The rendering object is attached to the object, or accessed if it already exists, using the *Add/Edit Rendering* button in the *Object Properties* dialog. It has the following attributes:

GradientType
The type of the gradient fill, which determines both the gradient geometry as well as the colors used for rendering. The following types of the gradient geometry are supported:

- LINEAR - a fill gradient with color changing along a line
- SPHERICAL - a fill gradient in the form of a sphere
- ELLIPTICAL - same as the SPHERICAL, but stretches with the object
- CONICAL - a fill gradient in the form of a cone

•LINE WIDTH - a line gradient for rendering 3D lines. The color gradient changes in the direction perpendicular to the direction of the line. Only the *Gradient Color* and *Gradient Length* attributes are used with the LINE WIDTH gradient.

The gradient color usage can be DIRECT (the color changing from object color to gradient color) or INVERSED (from gradient color to object color).

The LINEAR gradient can also be ACYCLIC (the color changing from the first color to the second color) or CYCLIC (from the first color to the second color and back to the first one).

The CONICAL gradient can be relative or absolute. For the relative conical gradient, the center is defined in relative coordinates as described below. For the absolute gradient, the center is defined using the world coordinates, which makes it possible to keep a constant center position when the object's shape changes.

If the value of the gradient type is NONE, the gradient is disabled.

GradientColor

The second color for the gradient fill. For polygons and polygon subclasses, the gradient fill renders the object with a color smoothly changing from the object's *FillColor* to the gradient color, according to the gradient type. For text objects, the color changes from the text's *TextColor* to *GradientGolor*. For polygons with no LINE FILL and the LINE WIDTH gradient, the color changes from *EdgeColor* to *GradientColor*.

GradientAngle

The gradient angle for linear gradient, the start angle for conical gradient. The angle is measured counter-clock wise relative to the X axis. For example, an angle of 0 with the linear gradient type results in a horizontal left-to-right gradient fill.

GradientLength

The relative length of the gradient in the range of 0 to 1, controlling the percentage of the object to be rendered with a gradient fill (the rest of the object will be rendered with a solid color). For example, a value of 1 results in the whole object being rendered with a gradient fill, with the gradient starting on one side of the object and ending on the other. If the value is equal to 0.5, half of the object will be rendered with a gradient fill, and the other half will be filled with a solid color. If the *Gradient Length* is larger than 1, the gradient fill will extend beyond the boundaries of the object and will be clipped.

GradientCenter

This G type attribute defines the center of the gradient fill in relative coordinates, so that the [0;1] range of each of its coordinates corresponds to the object's boundaries in the specified direction, with the direction of each coordinate coinciding with the corresponding axis. The Z coordinate of the center is ignored. For example, a value of (0.5, 0.5, 0.5) centers the gradient inside the object. A value of (0, 0, 0) positions the gradient's center in the lower left corner of the object. Values outside of the [0;1] range will position the center of the gradient outside of the object and result in the gradient being partially clipped. The attribute is ignored for the LINE WIDTH gradient.

GradientResolution

The number of segments used to render the gradient fill. Increasing this number will increase the rendering quality for the price of slower performance. For environments with OpenGL support (as well as the Java and C#/.NET versions of the Toolkit), gradient types other than conical are rendered natively and gradient resolution is ignored. The attribute is also ignored for the LINE WIDTH gradient.

On systems with a limited number of colors, the rendering quality is also limited by the number of colors available in the color table, and increasing the gradient resolution above a certain value will not further improve the rendering quality.

ShadowOffset

The cast shadow offset in pixels. The X and Y coordinates of the offset value define the shadow pixel offset in the corresponding direction. Negative values may be used to inverse the offset's direction. If the value is (0,0), no shadow is rendered.

The Z component of the offset value is interpreted as the shadow transparency. If the Z value is between 0 and 1, the shadow is rendered as transparent, with a greater transparency for the Z values closer to 0.

For environments with the OpenGL support (as well as the Java and C#/.NET versions of the Toolkit), the transparency is rendered as true alpha-blending. For the GDI versions of the Toolkit, the transparency is supported only in the Unix environment, where it is simulated using dithering patterns.
Transparency is not supported in the Postscript output.

ShadowColor

The color of the cast shadow.

FillDirection

The angle defining the direction of the fill dynamics. The angle is measured counter-clock wise, relative to the X axis. For example, a value of 0 results in a horizontal fill from left to right. In the Builder, the angle value may be entered in the *Attribute* dialog (ellipsis button ...), or one of the predefined values (UP, DOWN, LEFT, RIGHT) may be selected from the option menu.

FillAmount

The relative value in the range of 0 to 1 defining the percentage of the object filled with the object's *FillColor* (or *TextColor* for text objects). The *FillDirection* attribute defines the fill direction. If the value is 1, the whole object is drawn. If the value is less than 1, only a portion of the object's fill will be drawn, as defined by the fill amount. The object's fill type must have FILL enabled in order to be rendered. The object's edge is not affected by the value of the Fill Amount. For text objects, only the text is clipped; the text box is not affected.

ArrowType

The type of arrow(s) attached to a polygon or any of its subclasses (spline, arc or connector). The arrow type is a composite attribute that determines both the arrows' type (EDGE, FILL or FILL & EDGE) and position (START, END, START AND END or MIDDLE). For arrows positioned in the middle of the polygon, the arrow type also determines an arrow's direction: DIRECT (from start to end point) or INVERSED (pointing from end to start). If the value is NONE, no arrows are rendered. The difference between FILL and FILL & EDGE arrow types becomes visible for lines with line width greater than 1. The geometry of the arrowheads is controlled by the *ArrowShape* attribute and is adjusted to match the line width for thick lines.

The value of the attribute contains bytes: the low order byte defines the arrow position and the high order byte defines the arrow fill type, which may have the same values as the polygon fill type.

ArrowShape

The shape of the arrow in pixels. The X and Y coordinates of the arrow shape value are used. The X coordinate defines the length of the arrow along the line, the Y coordinate

defines the width of the arrow's one side in the direction perpendicular to the line.

The X and Y values of *ArrowShape* specify width and length of the arrow relative to the line width of the polygon. The actual width and length of the arrow increase proportionally when the line width increases. Negative values for both X and Y parameters may be used to define the absolute length and width of the arrow, so that the arrow's dimensions remain constant when the line width changes.
If null value is specified (0,0,0), the value of the *GlgArrowShape* global configuration resource is used (see *Appendices* on page 341 of the *GLG Programming Reference Manual*).

Delete Rendering
This button is present only in the Builder; it deletes the rendering object.

A rendering object may be reused by marking it with the *Mark* button at the top of the *Properties* dialog. Use the *Edit, Add or Use Marked Object, Rendering Attributes* menu option to add the marked rendering attributes to another object.

When a rendering object is added to all objects in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added rendering objects (the attributes are constrained if the default *Constrained Clone* setting is used). When a marked rendering object is reused, constraining of corresponding attributes of the new copy and the original rendering object is also controlled by the Builder's *Attribute Clone Type* option.

BoxAttributes

The *Box Attributes* object is used to keep attributes of an optional box drawn around text object. The box is drawn only if the Box Attributes object is attached to a text object. The size of the box is determined by the text object and is expanded automatically to fit the text's string. The box attributes object inherits most polygon attributes like *FillColor, EdgeColor,* and *FillType*, and has the following additional attributes:

BoxOffset
Pixel offset between the text and the box's edge. Only the X and Y coordinates of the G-type offset value are used, defining the pixel offset in the corresponding direction.

BoxEdgeColor
Box's edge color: a more specific name for accessing box's edge color.

BoxFillColor
Box's fill color: a more specific name for accessing box's fill color.

Delete Box Attributes
This button is present only in the Builder; it deletes the Box Attributes object.

A Box Attributes object may be reused by marking it with the *Mark* button at the top of the *Properties* dialog. Use the *Edit, Add or Use Marked Object, Text Box* menu option to add the marked box attributes to another text object.

When a Box Attributes object is added to all text objects in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added box attribute objects (the attributes are constrained if the default *Constrained Clone*

setting is used). When a marked Box Attributes object is reused, constraining of corresponding attributes of the new copy and the original object is also controlled by the Builder's *Attribute Clone Type* option.

Line Attributes

The *Line Attributes* object is used to keep attributes of lines and polygons used to render internal components of a chart or an axis. The Line Attributes object inherits most polygon attributes like *EdgeColor, LineWidth* and *LineType*. It may also contain *FillType* and *FillColor* attributes for filled polygons, and has the following additional attribute:

Opacity
Opacity in the range from 0 to 1. If set to 0, the line will be completely transparent.

A Line Attributes object may be reused by marking it with the *Mark* button at the top of the *Properties* dialog. Use options from the *Edit, Add or Use Marked Object* menu to add the marked line attributes to another object. The *Edit, Add or Use Marked Object* menu has several entries depending on the way the marked line attributes are used in the drawing: *Axis Tick, Chart Grid, Chart Cross-Hair, Chart Background* and *Plot/Level Line*.

When a Line Attributes object is added to all objects in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added Line Attributes objects (the attributes are constrained if the default *Constrained Clone* setting is used). When a marked Line Attributes object is reused, constraining of corresponding attributes of the new copy and the original object is also controlled by the Builder's *Attribute Clone Type* option.

Colortable

A colortable object defines a set of colors allocated for a viewport and provides an efficient way to manage colors for non-TrueColor visuals. On TrueColor systems, and in the Java and C#/.NET versions of the Toolkit, colortables are not used at run time. In the Builder, colortables are used even on the TrueColor systems to define the number of colors displayed in the Builder's color palette.

The color of an object in a viewport is defined by its three color coordinates, but the color that is actually displayed is the nearest neighbor in the color table to the point defined by those coordinates. This can yield surprising results in viewports with restrictive color tables on non-TrueColor systems.

Note that the total number of colors available is the number of colors defined by the *ColorFactor* attribute times the number of grades specified with *GradeHint*. If the product of the two numbers is greater than the number of colors your screen can display, the total number of available colors may be less than the number of colors in the color table. For example, you are limited to 256 different colors at any one time on an 8-bit color machine, and this hardware limitation takes precedence over the software color table definition. (Remember that the color limit also applies to other applications that may be running at the same time, and may have color tables of their own.) The results are not easily predictable if you try to define more colors than your machine can display.

The colortable object is created automatically every time a viewport is created and has the following attributes:

Type
Defines the type of the color distribution: STANDARD or RAINBOW. The STANDARD distribution allocates evenly distributed colors in the RGB color space. The RAINBOW distribution uses an algorithm that allocates colors in a rainbow-like palette and makes dithering nicer.

ColorFactor
Defines the number of colors for the STANDARD colortable or the number of hues for the RAINBOW colortable. If set to 0, a default value is used. A value equal to 1 may be used to simulate a greyscale or monochrome display. This is an index into a table of predefined values. For the STANDARD colortable, the values are:

Color Factors for Standard Color Table

ColorFactor	**Number of Colors**
0	256 (default)
1	256
2	8
3	64
4	256 (maximum)

The *ColorFactor* values of 0 and 1 are mapped to 256 colors instead of 1, because it does not make any sense to have a colortable with just one color. If *ColorFactor* is greater than 4, the number of colors is still limited to 256 on 8-bit color machines. The ColorFactor mapping for the RAINBOW color table is:

Color Factors for Rainbow Color Table

ColorFactor	**Number of Colors**
0	19 (default)
1	1
2	7
3	19
4	37
5	61
6	91
7	127
8	169

Color Factors for Rainbow Color Table

ColorFactor	**Number of Colors**
9	217

NumColors
A read-only attribute that contains the actual number of colors used.

GradeHint
Defines a number of color intensities for every color hue for the RAINBOW colortable. If set to 0 or 1, the default value of six is used. A value of two indicates that each color shall have two grades: black and full strength.

NumGrades
A read-only attribute that contains the actual number of grades used.

PatternFactor
Defines a number of dithering patterns used to render color intensities. Using dithering increases the number of possible color intensities without increasing the number of allocated colors. A value equal to 0 causes only one pattern to be used, disabling dithering. Like the *ColorFactor*, this attribute is an index into a table of predefined values:

Dithering Patterns

PatternFactor	**Number of dithering patterns**
0	1
1	4
2	16 (default)
3	64
4	256 (maximum)

NumPatterns
A read-only attribute containing the actual number of patterns used.

RenderingColor
Defines a color used for simulating a monochrome or a greyscale display. Everything is rendered in different intensities of this color. This attribute has an effect only if the number of colors is equal to 1.

ColorCorrection
Controls the color correction for the colortable. If it is set to YES, color intensities are corrected to produce a bigger number of light colors in the colortable. If it is set to NO, there are more dark colors. Color correction affects only the RAINBOW color distribution.

When a color table is added to a group of objects using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added color table objects (the attributes are constrained if the default *Constrained Clone* setting is used).

Font Table

The font table object controls the selection of fonts available for use in a viewport. The font table contains a list of font families, and each family contains a list of fonts of different font sizes. A *Text* object in a drawing is rendered using fonts from the font table of the drawing's viewport. The text object's *FontType* attribute defines a font family index in the font table, and the *FontSize* attribute defines the font size index within the font family.

When a viewport is created, it inherits a default font table. A font table has the following attributes:

NumTypes
Defines the number of font types in the font table. Each combination of font family, weight, and style corresponds to a different style. For example, Courier, Bold, Italic represents a type.

NumSizes
Defines the number of font sizes in the font table.

Fonts
A container object containing a list of font families. Each font family contains a list of fonts of different sizes. Each font is stored in a font object described below.

If a custom font table is not provided, GLG drawings use a default font table. A custom font table can be specified in one of the following ways:

- To assign a custom font table and **store it in the drawing**, click on *More* in the viewport's *Properties* dialog, then click on the *Add Font Table* button. See the *Editing a Font Table* section below for details.
- To specify a font table **stored in an external file** that can be shared between multiple drawings, use the *FontTableFile* attribute in a viewport's *Properties* dialog. See page 95 for more details.
- To set a custom font table as a **global default**, use the *GlgDefaultFontTableFile* global configuration resource described on page 345 of the *GLG Programming Reference Manual*. This custom font table will be inherited by all viewports that use a default font table.

The *GlgDefaultFontFile*, *GlgDefaultNumFontTypes* and *GlgDefaultNumFontSizes* global configuration resources described on page 345 of the *GLG Programming Reference Manual* may be used to define a list of fonts for the default font table using a plain text file. In the X Windows environment, each entry of the file may contain either a font name, or a comma-separated list of font names comprising a font set (for font sets, the *MultiByteFlag* attribute will be set to an appropriate value automatically). The fonts defined in the file will override the fonts defined in the default font table. The *GlgDefaultPSFontFile* global configuration resource may be used to specify the location of the file that contains the list of the PostScript fonts for the default font table. This method is inferior compared to the use of the *GlgDefaultFontTableFile* resource and is provided for compatibility with previous releases.

ADVANCED: the *GlgDefaultFontTable* global configuration resource can be used in a program to set the global font table default to an object ID of a loaded or created font table.

Editing a Font Table

A font table can be edited by changing the number of font types and font sizes, and editing the fonts defined in the font table. The dialog for editing fonts presents two lists: the list on the left displays font families, and the list on the right displays font sizes of the selected font family. The font *Properties* dialog on the far right displays properties of the selected font. A font browser for

selecting a font can be started by clicking on the ellipsis button ... for the font name attribute (*WinFontName* on Windows and *XFontName* on X Windows). Refer to the *Font* section below for more information.

It is the user's responsibility, when editing fonts in a table, to arrange them in such a way that all fonts in a font family have the same font type and the sizes range from the smallest for the first font to the largest for the last font. If the order is incorrect, text fitting for a SCALED text object may fail or yield odd results.

The font table's *Properties* dialog also contains buttons for reusing a font table. A font table may be reused by marking it with the *Mark* button at the top of the *Properties* dialog. Use the *Edit, Add or Use Marked Object, Font Table* menu option to add the marked font table to another viewport. The font table can also be reused by using the *Mark Font Table* and *Use Marked Font Table* buttons, or by saving it into a file and loading into another viewport by using the *Save Font Table* and *Load Font Table* buttons in the *Properties* dialog.

When a font table is added to all viewports in a group using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added font tables (attributes are constrained if the default *Constrained Clone* setting is used). When a marked font table is reused, constraining of corresponding attributes of the new copy and the original font table is also controlled by the Builder's *Attribute Clone Type* option.

Font

The font object is used to keep information about one font. It is created automatically every time a font table object is created and has the following attributes:

MultiByteFlag

Specifies whether the font handles the characters as single-byte, multi-byte or UTF-8. The attribute also controls the font usage as follows:

- The UTF8 setting is used for rendering strings with the UTF-8 encoding. In the X Windows environment, it uses fonts with the UTF-8 encoding . On Windows, it causes the wide character (Windows' UNICODE) version of the font to be used for rendering UTF-8 strings. If the UTF8 setting of the font and the rendered text string do not match, an appropriate encoding conversion will be automatically performed.
- In the X Windows environment the attribute also controls the use of font sets. If the value of the attribute is set to SINGLE_BYTE, the *XFontName* attribute specifies a single font to use. Any other value causes the value of the *XFontName* attribute to be interpreted as a comma-separated **font set** containing one or more fonts. All the text objects that use the font will be rendered using the specified font set via the Xmb family of the text rendering functions.

For all fonts in the default font table, as well as fonts with the font charset attribute set to DEFAULT_CHARSET on Windows, the actual value of the flag is determined automatically based on the system locale. The *GlgMultibyteFlag* global configuration variable may be used to specify the value of *MultiByteFlag* that overrides the automatic setting for these fonts.

The attribute value is ignored in the Java and C#/.NET versions of the Toolkit.

Note: The multi-byte setting does not mean that each character has the same fixed length of more than one byte. Instead, it means the use of variable width characters where each character may consist of one or more bytes.

XFontName
Holds the name of the font to use when the drawing is displayed in the X Windows environment. If wild card characters are used, the first matching name is used. If *MultiByteFlag* is set to UTF8, the font with the UTF-8 encoding must be used.

If the font name starts with the '$' character, the rest of the string after the '$' character defines the name of the environment variable that specifies the font name to use. This could be used as an escape mechanism for defining a special font at run time.

WinFontName
Holds the name of the font to use when the drawing is displayed in the Microsoft Windows environment. If *MultiByteFlag* is set to UTF8, the font with the Windows UNICODE encoding must be used. The environment variable escape mechanism described for *XFontName* is supported for *WinFontName* as well.

WinCharset
Specifies the font's charset. If the value of the attribute is set to DEFAULT_CHARSET, the font with the charset of the current system locale will be used, otherwise the font with the specified charset will be chosen.

For fonts whose charset attribute is set to DEFAULT_CHARSET (including the fonts of the default font table) the attribute may be set globally by setting the *GlgFontCharset* global configuration variable.

The attribute value is ignored in X Windows environment, in the Java and C#/.NET versions of the Toolkit, and for fonts with *MultiByteFlag* set to UTF8 on Windows.

JavaFontName
Holds the name of the font to use when the drawing is displayed in the Java environment.

FontName
A cross-platform alias for the font name attribute. It is dynamically mapped to one of the above attributes at run time depending on the environment. For example, if the Java environment is used, the *FontName* resource name may be used at run time to access the *JavaFontName* attribute.

PSName
Specifies the name of the PostScript font to use in place of the font when producing PostScript output.

By default, the PostScript Font Name attribute is set to match the Font Name attribute, which is sufficient for Times, Courier and Helvetica families of fonts. For other fonts or if a different font mapping is desired, it has to be set manually.

Light Object

The light object is used by a viewport to hold the viewport's lighting attributes. The light object has the following attributes:

LightType

Specifies a type of shading to be used in rendering three-dimensional objects. Currently available values are:

NONE (GLG_NO_LIGHT)

All polygons will be rendered in their original colors regardless of the orientation (as long as the *LightCoef* and *AmbientCoef* attributes add to one). The coloration of a polygon surface always depends on the total light intensity (sum of the *LightCoef* and *AmbientCoef* attributes), and its *FillColor* attribute.

FLAT (GLG_FLAT_LIGHT)

The actual color used to fill a polygon also depends on its position relative to the light vector (directed from *LightPoint* to *LightDirection*). The light is infinitely distant and the light rays are parallel to one another.

POINT (GLG_POINT_LIGHT)

The actual color used to fill a polygon also depends on its position relative to the light vector. The position of the light source is defined by the *LightPoint* attribute.

Note: The POINT light setting is enabled only when the OpenGL driver is used. If the GDI driver is used, it behaves as the FLAT light option.

LightPoint and **LightDirection**

Define start and end points of the light vector. Light is directed from the *LightPoint* point to the *LightDirection* point. Note that these two points define the direction of the light source. The light itself appears to be infinitely distant.

LightCoefficient

Controls the proportion of a viewport's light cast by the light source at *LightPoint*. This is a scalar value, ranging from 0 to 1. The sum of the light coefficient and the ambient coefficient should be between 0 and 1, otherwise color distortion may occur. (Of course, setting the sum greater than 1 may be used to produce special effects.)

AmbientCoefficient

Defines the proportion of a viewport's light cast by ambience. In a sense, this controls how bright the completely shaded places are. The coefficient is a scalar value, which can range from 0 to 1.

A light object may be reused by marking it with the *Mark* button at the top of the *Properties* dialog. Use the *Edit, Add or Use Marked Object, Light Object* menu option to add the marked light object to another viewport.

When a light object is added to a group of viewports using the group's *Edit All* option, the *Attribute Clone Type* option of the Builder controls constraining of corresponding attributes of the added light objects (the attributes are constrained if the default *Constrained Clone* setting is used). When a marked light object is reused, constraining of corresponding attributes of the new copy and the original light object is also controlled by the Builder's *Attribute Clone Type* option.

A polygon's *Shading* attribute provides additional control over shading of individual polygons in the drawing.

For more information about lighting, see the *Lighting* section on page 42 of the *Structure of a GLG Drawing* chapter.

Transformation Object

The transformation object describes a transformation associated with a GLG object. A special type of a transformation object is also used to implement alarms.

The lists in the following sections describe the attributes of the GLG transformation objects. The default names of these attributes are *XformAttr1* through *XformAttr6*. The attributes below are listed in the order given by their default names, but we have used more descriptive names to help explain their use. These names are also used in the GLG Graphics Builder.

Stock Transformations vs. Predefined Dynamics

There are two sets of dynamics options for object attributes: the **stock transformations** and **predefined dynamics**. The stock transformations are basic transformation types used as building blocks to implement dynamic behavior. Predefined dynamics are pre-created combinations of stock transformations that provide easy to use options for the most common dynamic actions in the GLG editors. Predefined dynamics may also be used by system integrators to extend GLG editors with elaborate application-specific dynamics.

Predefined dynamics represent a collection of several stock transformations wired together to implement a specific dynamic behavior. Most of the parameters of the transformations used to implement the predefined dynamics are hidden from the user, and only the essential parameters marked as public are presented to the user as a simple list of public properties. The *Options, Dynamics Options* menu of the Graphics Builder contains options that control how the predefined dynamics are displayed in the Builder's dialogs.

The following chapters list the available types of the stock transformations first, and list the predefined dynamics choices at the end.

Geometrical Transformations

The geometrical transformations are those that can be applied to a point in three-dimensional space, or a data point of type G. Because graphical objects are described by a set of three-dimensional control points, these transformations can, by extension, be applied to entire three-dimensional graphical objects. (For information about the distinction between attaching a transformation to an object and to the object's points, see the *Transformations as Objects* section on page 36.)

Several geometrical transformations can be concatenated together into a list, with each transformation applied to the output of the previous transformation in the list.

When the rendering routines are attempting to draw a graphical object, they first look to see what transformations are attached to the object. If there are any, they are applied to the object before it is displayed.

The set of transformation objects available in GLG can be divided into matrix and parametric. The matrix stored by a matrix transformation may be directly applied to the graphical objects, while the parameters in a parametric transformation must first be used to create the transformation matrix before it can be applied. While the matrix transformations store information in a more compact way,

they are static and can not be easily changed. The parametric transformations can be changed dynamically by changing the transformation's parameters. For more information about the structure of GLG transformations, refer to the *Transformations* section on page 35.

As a convenience, most of the parametric transformations include a scaling factor with their most commonly used parameters. This is to say that these parameters are actually the product of two attributes. For example, the distance moved in a MoveBy transformation object is the product of the *Distance* and the *Factor* attributes. A user might want the actual distance to range between 0 and 500 units, while the input data received is a number between 0 and 1. He or she could set the *Distance* attribute to 500, and use the resource-setting mechanism to control the *Factor* attribute with the actual data received. This also separates the input logic from the geometry of the drawing, as the *Distance* may be changed without affecting the input data. If the input data is in a range different from 0 to 1, a range transformation may be attached to the *Factor* attribute to handle input data in an arbitrary range.

Note that though the following sections describe the entire set of GLG transformation objects, a program can create more specialized or elaborate transformations simply by querying a drawing's control points, calculating new values within the program, and changing those resources. It is best to think of the list of transformations that follow as a list of those transformations for which additional programming is not necessary, rather than as a complete list of what is possible.

Matrix

The *Matrix* transformation stores a 4x4 matrix with which to transform a geometrical value. This is the standard matrix transformation used in computer graphics, and the derivation is available in many texts on that subject. To use the GLG Toolkit, you need only understand that the matrix, when multiplied by a point in three-dimensional space, produces the coordinates of another such point. The mapping of input points onto output points by this matrix multiplication is a transformation. The matrix itself is the only attribute of a matrix transformation object. Its name is *Matrix*, and it is a read-only attribute. This is to say that once the matrix object has been created, it may not be edited directly, although it may be replaced.

Partly because of the read-only nature of the *Matrix* attribute, and partly because of the unwieldy nature of a matrix value, the matrix transformation is often called a **static** transformation. All the other geometrical transformation objects are **dynamic** objects, since they simply store the values necessary to create a transformation matrix on the fly. These values may be edited, and accessed, as resources like any others.

MoveBy

The *MoveBy* transformation object moves the input geometrical value a given distance along the X, Y, or Z axis. It has three attributes:

Direction
Specifies the direction of the motion, selected at the creation time. The available values are X, Y, or Z, or XYZ.

Distance
A scalar (double-precision) specifying the distance to move.

Factor
Normalized move parameter. The actual distance moved is the product of the *Factor* and the *Distance* attribute.

If the *Direction* attribute has the *XYZ* value, there are three *Distance/Factor* pairs, one for each dimension.

Move

The *Move* transformation, like the *MoveBy* transformation, is simply used to relocate a geometrical value. The difference is in how the move is specified. A *MoveBy* transformation moves a point a given distance in a predetermined direction. A *Move* transformation specifies the direction explicitly by using two points in the drawing.

The Move object has the following attributes:

StartPoint
The start point of the move vector, specified with a geometrical value.

EndPoint
The end point of the move vector, specified with a geometrical value

Factor
Normalized move parameter. The actual move vector is the product of the *Factor* and the vector defined by the *Start* and *End* points.

Rotate

The *Rotate* transformation specifies a point in space to rotate about, and a number of degrees to rotate.

Rotation Axis
Specifies the axis around which the object rotates and is selected at the creation time. The available values are X, Y, or Z.

Center
The center point, specified as a value.

Angle
Rotation angle in degrees counter-clockwise.

StartAngle
Start angle of rotation in degrees, measured counter-clockwise. The object is always rotated from its original position by the start angle even if the factor is 0. The start angle is convenient for defining the start position of rotation without actually rotating the object.

Factor
Normalized rotation parameter. Changing the factor from 0 to 1 changes the actual rotation angle from *StartAngle* to *StartAngle* + *Angle*.

Shear

The *Shear* transformation is like a rotation, except that each point of the shape being sheared is constrained to stay on a line parallel with the shear direction.

Direction
Specifies the direction of the shear and is selected at the creation time. Possible values are X, Y, and Z.

Center
The center point, specified as a geometrical value.

Shear
Scalar shear coefficients relative to the selected shearing axes.

Factor
Normalized shear parameter. The actual shearing coefficients are products of the *Factor* and the *Shear* coefficients.

Scale

The *Scale* transformation defines a center point and a factor. An input point is transformed by measuring its distance from the center point, and moving it along the line defined by those two points to a point whose distance from the center point is the original distance times the *Scale* value.

Center
The center point, specified as a geometrical value.

Scale
Scalar scale coefficient.

Start Scale
Start scale coefficient.

Factor
Normalized scale parameter. Changing the factor from 0 to 1 changes the object's actual scale factor from Start Scale to Scale.

There is no separate **Mirror** transformation object. You can create one in the editor, but it is just a convenience, like the rectangle. A mirror transformation is simply a scale transformation with the *Scale* attribute set to -1.

In addition to the simple scale transformation object, there are corresponding *ScaleX*, *ScaleY*, and *ScaleZ* transformation objects, each of which act only upon the indicated dimension. The type of the scale transformation is selected at the creation time.

Path

The *Path* transformation moves a point along a predefined path polygon.

Path
Array of the path points. This is a group object that contains points of the path.The points may be edited, added or deleted.

Factor
This is a scalar value, specifying the current position as a distance along the polygon's perimeter. The first defined point on the polygon is at *Factor=0*, and the last is at *Factor=1*.

Rotate Flag
Controls the way object rotates as it follows the path. The flag may have the following values:

DON'T ROTATE
Object does not rotate.

ROTATE

Object rotates to match the tangent of the path in the current position. This setting may be used to create an object that will rotate as it moves along a curved path.

ROTATE NO ORIGIN

Advanced setting, same as rotate, but the origin parameter is ignored. The object is not moved to the beginning of the path as defined by the origin parameter. The effect of this setting is visible only if the origin was moved to a position different from the beginning of the path.

Origin

A point that defines how the object must to be moved to the beginning of the path when the path factor is 0. The object is moved by the vector defined by the *Origin* as the start point and the first point of the path as the end point. By default, the *Origin* is constrained to the first point of the path and the object is not moved to the beginning of the path. To move the object, unconstrain the *Origin* and then define it's new value. For example, placing the origin in the center of the object will cause the object's center to be aligned with the path as the object moves along it. You can also constrain the *Origin* to some point of the object, which will permanently align that object's point to the path.

Discrete

If set to YES, changes the behavior of the transformation by interpreting the value of *Factor* as an integer 0-based point index. Setting *Factor* to 1 moves the object from the first to the second point of the path; setting it to 2 moves the object to the third point, and so on. The factor value of 0 corresponds to the first point and returns the object to the beginning of the path.

Offset

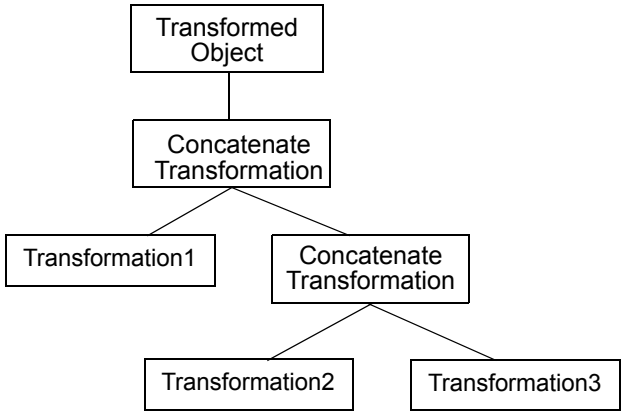
Defines an initial path offset which is added to the value of the *Factor*.

Wrap

If set to YES, the values of the *Factor* outside of the range ([0;1] for non-discrete path transformations, or from 0 to the number of path points for discrete paths) get "wrapped around" instead of being truncated.

Concatenate

The *Concatenate* transformation is used to build a list of geometrical transformations attached to a single object. This transformation object has only two attributes, that point to the first transformation on the list and the second. If more than two transformations must be attached to an object, then multiple Concatenate objects can be used. For example, to hold a list of three transformation objects, use two *Concatenate* transformations, as in the following diagram:



Using the Concatenate Transformation Object

To add another transformation to the list, replace *Transformation3* with another concatenate transformation object, which holds *Transformation3* as the first attribute and the new transformation as the second attribute.

Note that the GLG Graphics Builder does not explicitly manipulate concatenate transformation objects. The Builder uses concatenate transformation to build and edit lists of transformations attached to an object, but the overhead of keeping track of the concatenate objects is hidden from the user.

The concatenate transformation object has just two attributes:

Xform1
The first transformation object.

Xform2
The second transformation object.

World Offset

The *World Offset* transformation may be attached to a point to maintain a constant offset from another point. This transformation may be attached only to an object's control point and not to the object itself. It has the following attributes:

Anchor Point
Specifies the coordinates of the anchor point. This attribute can be constrained to another point to use that point as an anchor.

X Offset
Offset along the X axis in the world coordinates.

Y Offset
World offset along the Y axis.

Z Offset
World offset along the Z axis.

To add the world offset transformation, use *World X, World Y* or *World XY* offset type options in the *Add Dynamics* dialog. All of them add the same type of the world offset transformation, the difference is that they initialize unused offsets to 0 for convenience.

Moving a point with the worlds offset transformation modifies *X, Y* and *Z offset* parameters of the transformation instead of changing the point's coordinates.

Screen Offset

The *Screen Offset* transformation is similar to *World Offset*, but allows the user to define an offset in either the world coordinates or screen pixels. If a pixel offset is used, the transformation maintains a constant pixel offset when the drawing is resized; however, the offset is still a subject to other transformations and will change together with the drawing when the drawing is zoomed or the object is scaled.

Same as the *World Offset*, this transformation may be attached only to an object's control point and not to the object itself. It has the following attributes:

Anchor Point
Specifies the coordinates of the anchor point. This attribute can be constrained to another point to use that point as an anchor.

X Offset
Offset along the X axis.

Y Offset
Offset along the Y axis.

Z Offset
Offset along the Z axis.

X Offset Type
Units of the *X Offset*: world coordinates (WORLD) or pixels (SCREEN).

Y Offset Type
Units of the *Y Offset*: world coordinates (WORLD) or pixels (SCREEN).

Z Offset Type
Units of the *Z Offset*: world coordinates (WORLD) or pixels (SCREEN).

The *Screen Offset* transformation allows the user to mix a world offset in one direction and pixel offset in another by using different offset types for different axes.

To add the screen offset transformation, use *Pixel X,* the *Pixel Y* or the *Pixel XY* offset type options in the *Add Dynamics* dialog. All of them add the same type of the screen offset transformation, initializing unused offsets to 0 for convenience.

Moving a point with the screen offset transformation modifies *X, Y* and *Z Offset* parameters of the transformation instead of changing the point's coordinates. The screen offset transformation is not allowed inside the GIS object, except when it is used inside icons implemented as reference objects of a fixed size.

Screen Scale (ADVANCED)

The *Screen Scale* transformation is similar to the *Scale* transformation, but it automatically adjusts its scale factor to maintain an object's dimensions constant in the selected direction when the drawing is resized. The transformation is used by some of the graphs for implementing desired layout behavior and has the following attributes:

Center
The center point, specified as a geometrical value.

Factor
Scalar scale coefficient. The actual scale factor is a product of the Factor and the automatically adjusted scale factor used to compensate the change of the window size.

To add the screen scale transformation, use *ScaleScrX, ScaleScrY* or *ScaleScrZ* scale type options in the *Add Dynamics* dialog.

Scalar Transformations

There are several transformation objects designed to apply to simple scalar (D) values; these transformations can be attached to attributes of D type, such as *LineWidth* or *Visibility.* A scalar transformation takes an input value, transforms it according to the type of the transformation and assigns the resulting output to the attribute's transformed value. What is used as the input value depends on the type of the transformation.

Some scalar transformations **use the value of the attribute** they are attached to as the input value, and set the transformed value of the attribute to the output. For example, the *Divide* transformation takes the value of the attribute, divides it by the transformation's *Divisor* parameter and sets the transformed value of the attribute to the resulting output. In the Graphics Builder, the attribute's value and the transformed value are shown in the *Attribute* dialog as *Value* and *XfValue*; the attribute's text field in the *Properties* dialog shows the attribute's *Value* and allows in-place editing.

Other scalar transformations **ignore the value of the attribute** and use one or more transformation parameters as input values. For example, the *Range Conversion* transformation ignores the value of the attribute, uses its *Input Value* parameter as an input and sets the transformed value of the attribute to the resulting output. In the Graphics Builder, if an attribute that has a transformation that ignores the attribute's value, the *Value* field in the *Attribute* dialog will be disabled; the *Property* dialog will display the attribute's transformed value *(XfValue)* which also will be disabled for editing. Any changes of the attribute should be done by changing the input value of the attached transformation.

Scalar transformations cannot be concatenated, so you can't attach more than one scalar transformation to an attribute. However, you can "chain" transformations by attaching other scalar transformations to the parameters of a scalar transformation.

Refer to the *Common Attribute Transformations* section on page 166 for additional transformations that can be attached to scalar attributes.

Range Conversion

The *Range Conversion* transformation maps a range of input data values into a different range of output values. For example, if the transformation is set up to map the input values 12 through 20 to the values 0 to 1, an input value of 16 would produce an output value of 0.5.

By default, the two sets of bounds set up the mathematical relation between input and output; they do not impose limits. That is, the input bounds merely set the ratio of input to output. An input value outside the given bounds is mapped to a comparable output value outside the output bounds. That is, if the range transformation is set up to map the interval from 0 to 1 to the output interval 100 to 200, than an input value of 5 will map to an output value of 600. Similarly, the low and high bounds of the input range can be flipped, with the low higher than the high. In this case, the mapping, too, will be flipped. The transformation's *Truncate* parameter may be set to force the values inside the bounds.

The transformation ignores the attribute's *Value* and uses the transformation's *Input Value* parameter as input. The converted output value is assigned to the attribute's transformed value (*XfValue*). The attributes for this transformation are as follows:

In Low
The lower bound of the input data range.

In High
The upper bound of the input data range.

Out Low
The lower bound of the output data range.

Out High
The upper bound of the output data range.

Truncate
If set to YES, the output value outside the *OutLow*-*OutHigh* range will be adjusted to fit inside the range.

InputValue
The input value to be converted to a new range.

Backward Compatibility Note: A *Range* transformation was used in releases prior to 3.4. The *Range* transformation had the same parameters as *Range Conversion*, except for *Input Value*: instead, it was using the attribute's Value as input. The *Range* transformation was deprecated, but it is still supported for backward compatibility with old drawings.

RangeCheck

The range check transformation sets the attribute's transformed value (*XfValue*) depending on the input value being inside or outside of the specified range. The transformation may be used to activate blinking when the value goes out of range. The transformation ignores the attribute's *Value* and uses the transformation's *Input Value* parameter as input.

The attributes for this transformation are as follows:

High High
The *high high* input data range.

High
The *high* input data range.

Low
The *low* input data range.

Low Low
The *low low* input data range.

Input Value
The input value.

Equal Flag
Controls how input values equal to the range limits are handled. When set to ALARM_ON_EQUAL, input values equal to the limits are handled as alarms. When set to NO_ALARM_ON_EQUAL, values equal to the limits are handled as normal.

If the input value is outside of the *HighHigh-LowLow* range, the output value is set to 2. If the input value is outside of the *High-Low* range, the output value is set to 1. If the input value is within the *High-Low* range, the output is set to 0. To disable the *HighHigh-LowLow* check, set both of these attributes to 0.

Timer

The *Timer* transformation periodically changes a value of a scalar attribute, which can be used to implement various types of blinking or to perform run-time animation. The output value of the timer transformation is multiplied by the attribute's *Value* before being assigned to the attribute's transformed value (*XfValue*).

The timer transformation has the following attributes:

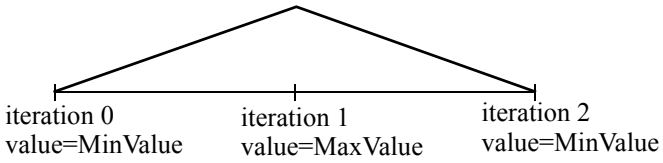
Update Type
The type of periodic updates to perform, may have one of the following values:
SAWTOOTH

A linear update type where the value gradually increases from the minimum to maximum value and then jumps back to the minimum:
TRIANGLE

A linear update type where the value gradually increases from the minimum to maximum value and then gradually decreases back to the minimum.
CIRCULAR
An update type for animating rotating objects. It is similar to the SAWTOOTH update type, except that the maximum value is never reached: instead, the value jumps back to the minimum value. This is used for animating rotational angles with the 0-360 degrees interval and makes sure that the rotating object does not "stutter" at the beginning and end of each of the rotational revolution, where the angle value of 360 degrees produces the same result as the value of 0 degrees.
SINE
A update using sine function.

Period
The number of value intervals it takes to change the value from the minimum to maximum value and back. The actual number of iterations for the whole period is bigger by one. For example, for the default period value of 2 the value alternates between the minimal and maximum value. The complete period takes 3 iterations: value=*MinValue* (iteration 0), value=*MaxValue* (iteration 1), value=*MinValue* (iteration2), but the number of intervals is equal to two (see the picture below)

.



The period may be set to a positive or negative value. The sign of the period value defines the timer's behavior when it is disabled. The detailed description is provided below.

Interval
The interval of periodic updates in seconds (default value is 1 second).

MinValue
The minimum data value.

MaxValue
The maximum data value.

Enabled
Disables the timer if set to 0, enables the timer if set to 1.

The timer is active only at run-time or in the prototyping mode. It is disabled in the editing mode for convenience. Internally, the timers are cached for efficiency and only one native timer is used for each collection of timers with the same period. This also synchronizes blinking of objects with the same timer intervals. When the drawing is initially loaded or reset, the timer always starts with the minimum value.

For timers with a positive *Period,* the timer is always set to the minimum value (*MinValue*) when disabled. For inversed behavior, simply switch the maximum and minimum values. For example, if MinValue=1 and MaxValue=0, the timer will start from 1 and will also stay at this value when disabled. When the timer is enabled again, it will continue updating in sync with the other timers that have the same period and are not disabled. This synchronizes blinking of objects with the same timer intervals after the timer was disabled and then enabled again.

For timers with a negative *Period*, the timer keeps its current state when disabled, instead of resetting to the minimum value. When the timer is enabled again, it will continue from the state where it was stopped, which may not match the state of other timers with the same period value.

Divide

The *Divide* transformation divides the attribute's *Value* by the value of a scalar divisor and assigns the result to the attribute's transformed value (*XfValue*). Its only attribute is the divisor itself, called *D Parameter.*

Linear

The *Linear* transformation has six attributes called *M*, *A, X, B, Y,* and *D*. The transformation ignores the attribute's *Value* and sets the attribute's transformed value (*XfValue*) to the result of the following expression:

M * (A * X + B * Y) / D

where *M, A, X, B, Y* and *D* are the value of the six transformation attributes.

Compare

A *Compare* transformation ignores the attribute's *Value* and sets the attribute's transformed value (*XfValue*) to the result of the comparison of two input parameters of the transformation. If the result of the comparison is True, the output value is set to 1, otherwise the output value is set to 0. The transformation has the following attributes:

OP

The comparison operation, may be one of the following:

==
!=
<
<=
>
>=

A and B Parameters

The input parameters to be compared.

Boolean

A *Boolean* transformation ignores the attribute's *Value* and sets the attribute's transformed value (*XfValue*) to the result of the boolean function of three input parameters of the transformation. If the result of the boolean function is True, the output value is set to 1, otherwise the output value is set to 0. The transformation has the following attributes:

Function

The boolean function, may be one of the following:

A || B || C
A || !B || C
A || B || !C
A || !B || !C
!A || B || C
!A || !B || C
!A || B || !C
!A || !B || !C
(A || B) && C
(A || B) && !C
(A || !B) && C
(A || !B) && !C
A && B && C
!A && B && C
A && !B && C
A && B && !C
!A && !B && C
!A && B && !C
A && !B && !C
!A && !B && !C

A && B || C
A && !B || C
A && B || !C
A && !B || !C
!A
A

A, B and C Parameters
The input parameters of the boolean function.

Boolean Converter (Bool(var))
The type of boolean conversion function used to convert double input values to boolean, may be one of the following:

var != 0. - True if the input value is not zero
var > 0. - True if the input value is positive
var > 0.5 - True if the input value is greater than 0.5
ABS(var) > 0.5 - True if the absolute value of the input value is greater than 0.5

Bitmask

A *Bitmask* transformation ignores the attribute's *Value* and sets the attribute's transformed value (*XfValue*) to the value formed by interpreting the states of four input parameters as bits of a 4-bit integer (from 0 to 15). A value of each input parameter is converted to a boolean *True* or *False*; if the result is *True*, the corresponding bit in the 4-bit integer output is activated. For example, the following combination of input parameters yields the output value of 13:

Bit 3 = 1
Bit 2 = 1
Bit 1 = 0
Bit 0 = 1

If input parameters are binary signals that represent a state of a device, a bitmask transformation can be used to change an object's color depending on the combination of the input signals. This can be accomplished by attaching a bitmask transformation to an the *Index* attribute of a list transformation attached to an object's *FillColor*, and arranging a list of colors to handle all combinations of inputs, depending on the number of used input parameters (2, 3 or 4).

The transformation has the following attributes:

Bit 3
Bit 2
Bit 1
Bit 0
The input parameters of the transformation.

Boolean Converter (Bool(var))
The type of boolean conversion function used to convert double input values to boolean, may be one of the following:

var != 0. - True if the input value is not zero
var > 0. - True if the input value is positive
var > 0.5 - True if the input value is greater than 0.5

ABS(var) > 0.5 - True if the absolute value of the input value is greater than 0.5

String Transformations

There are four string transformations that can be applied only to string attributes. All string transformations ignore the attribute's *Value* and assign the transformation's output to the attribute's transformed value (*XfValue*).

String transformations cannot be concatenated: only one string transformation can be attached to a string attribute. However, string transformations can be chained by attaching a string transformation to an attribute of another string transformation.

Refer to the *Common Attribute Transformations* section on page 166 for additional transformations that can be attached to string attributes.

String Formatting (Format S)

The string formatting transformation formats the string value of the *Data* attribute according to the formatting instructions in the *Format* attribute.

Format
A character string specifying how the *Data* attribute value is to be displayed. The format is specified with the standard C language *printf* format for strings, for example: *"String=%10s*". Using formats other than the variants of the *%s* format is not allowed and may result in a crash.

Escape sequences may be used to define native platform-specific formats for Windows and Unix platforms, as well as Java and C#/.NET environment, as described in the the *Scalar Formatting (Format D)* section below.

Data
The string data value to write into the text string.

Scalar Formatting (Format D)

The scalar formatting transformation formats the scalar value of the *Data* attribute with the formatting instructions in the *Format* attribute. If you constrain the *Data* attribute to the output value of a slider, you can produce a real-time display of the current slider value.

Format
A character string specifying how the *Data* attribute is to be displayed. The format is specified with the standard C language *printf* format. The type of the format specification must match the *Format Type* attribute. For example, the following format can be used to display a double value: *"Value=%.2f"*.

The following formats are supported:

double formats: %f, %F, %g, %G, %e, %E
integer formats: %d, %x, %X, %o

Using format types that do not match the *Format Type* attribute is not allowed and may result in a crash.

Escape sequences may be used to define native platform-specific formats for Windows and Unix platforms, as well as Java and C#/.NET environment. The platform-specific formats may be specified by surrounding them with HTML-style brackets:

```
<platform>native format</platform>
```

where platform may be one of the following: *java*, *c#*, *c_unix* or *c_windows*. One or more platform-specific formats may be specified before the generic platform-independent format that will be used for the remaining platforms. For example, the following format uses different native format specifications for C#, Java and C/C++/ActiveX environments:

```
<c#>{0:N2}</c#><java>%,2f</java>%.2f
```

In Java, the native format is passed as a *format* parameter to the *format* method of a *Formatter* object. In C# it is passed as a *format* parameter to the *String.Format* method for double, integer and string formats, and as a *format* parameter to the *ToString* method of a *DateTime* object for time and data formats.

Data
The scalar data value to write into the text string.

Format Type
The type of the format to use: DOUBLE or INTEGER. If an integer format is used, the data value is first cast to an integer type, with no rounding performed.

Time Format

The time formatting transformation formats the supplied time value with the requested time format.

Time Format
A character string specifying a desired time format (*"%X %x"* by default). Refer to the description of the axis object's *Time Format* attribute on page 126 for information on the supported time formats.

Escape sequences may be used to define native platform-specific formats for Windows and Unix platforms, as well as Java and C#/.NET environment, as described in the the *Scalar Formatting (Format D)* section above.

MilliSec Format
Specifies a double-precision C-style format for an optional display of fractional seconds at the end of the time string in the form of milliseconds. For example, *".%03.0f"* will display 270.5 milliseconds as *".270"*. It may be set to an empty string to suppress milliseconds display.

Time Input
Supplies the time to be displayed measured as the number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC. When the drawing is loaded (or reset), the attribute value is set to the current time and then stays constant until the value of the attribute is set using *GlgSetDResource*. To display current time, use a negative value: the display will be updated with the current time every time the property is set to a negative value. To automatically update display with current time, use the *Time Display* and *Date Display* predefined transformations described on page 173.

Time Display
Specifies how to interpret the value of the *Time Input* attribute. If set to RELATIVE, the value of the *Time Origin* attribute will be subtracted from *Time Input* to display time interval elapsed since the *Time Origin*. If set to LOCAL or UTC, the time will be displayed as a local or UTC time, respectively.

Time Origin
Specifies the start time to be subtracted from *Time Input* to display relative time. Refer to the description of the axis object's *Time Origin* parameter on page 130 for examples of relative time usage.

String Concatenation

The string concatenation transformation replaces the transformed string with the concatenation of the substrings provided as transformation parameters. The string concatenation transformation may be used to create a text object that displays a label, value and units, each controlled by a separate parameter. A *Format D* transformation may be attached to one of the substring parameters of the transformation to display a numerical value as a string. The transformation has the **A, B, C, D,** and **E** parameters that specify strings to be concatenated.

Common Attribute Transformations

Common attribute transformations can be attached to an attribute of any type: D, S or G. These transformations ignore the attribute's *Value* and assign the transformation's output to the attribute's transformed value (*XfValue*). The available types of the common attribute transformations are listed below.

List

The *List* transformation uses an integer index to select an attribute value from a list. For example, a list of colors may be attached to the *Fill Color* attribute of an object and the list's *Index* used to control what color is displayed: the index of 0 will display the first color, the index of 1- the second color, and so on.

The list transformation may be attached to attributes of any type (D, S or G). For example, you can attach a list of strings to be displayed in a text object via the text's *String* attribute, or attach a table of line widths for the polygon's *Line Width* attribute.

The attributes for this transformation are as follows:

List of Values
The list of attribute values to use, which may be edited.

Index
The zero-based index controlling which value from the list is displayed.

For color attributes with the list transformation attached, the transformed color value is calculated by adding the color value specified in the list to the value of the attribute. When the transformation is attached in the Builder, the first color in the list is set to the attribute's color, and the attribute's color value is set to a black color (0,0,0) to prevent interference.

SList

The *SList* transformation is similar to the list transformation, except that it uses a string as an input variable instead of a list index. The value of the string input is compared with the entries in a provided list of strings, and if it matches, the corresponding entry in the list of values is used as the output. If no match is found, the value of the last entry in the list of values is used. Same as the list transformation, the SList transformation may be attached to attributes of any type (D, S or G).

The *SList* transformation has the following attributes:

List of Values
The list of attribute values to use. It may have one more entry than the list of strings: this last entry is used when no match is found.

List of Strings
The list of strings to compare the input string with.

String Value
The input string controlling which value from the list of values is displayed.

Threshold

The *Threshold* transformation compares an input scalar value with the list of thresholds and selects an attribute value from a matching list of values. For example, a threshold transformation containing a list of two threshold values and a list of three colors may be attached to the *Fill Color* attribute of an object and the threshold's *Value* used to control what color is displayed: the value less than the first threshold value will display the first color, the value between the first and the second threshold value - the second color, and the value bigger than the second threshold - the third color. Notice that the number of colors is one bigger than the number of threshold values.

The threshold transformation may be attached to attributes of any type (D, S or G). For example, the list of attribute values may be a list of strings to be displayed in a text object's *String* attribute, or a list of line widths for the polygon's *Line Width* attribute.

The attributes for this transformation are as follows:

List of Values
The list of attribute values to use, which may be edited. The number of values in the list must be bigger than the number of the threshold values by 1.

Thresholds
The list of threshold values to use, which may be edited. Since a threshold list is processed sequentially using *LESS THAN* or *LESS OR EQUAL* comparison, each threshold in the list must be bigger than the previous one.

Value
The value that, after mapping to the list of thresholds, controls which value from the list of attribute values is displayed.

Equal Flag
A flag that controls the threshold comparison mode. It may have values of *LESS* (use i^{th} attribute of the *List of Values* if the value is less than i^{th} threshold) or *LESS OR EQUAL*.

Transfer

The *Transfer* transformation is used to transfer a value from one attribute object to another. It can be used to implement a "one-way" constraint, where changes in one object affect another, but not vice versa. The transfer transformation may be attached to an attribute of any type (D, S or G).

The transfer transformation has the following attributes:

Source
Constrain this attribute to the "source" attribute. Its value is passed into the *Buffer* attribute.

Buffer
If you want to attach a transformation to this transfer, attach it to this attribute. The transformation will alter the transferred value without affecting the source value.

Use Value
If set to *Value*, the *Value* of the *Source* attribute before applying any attribute transformations is used. If set to *XfValue (*default), the transformed value of the *Source* attribute is used. The transformed value includes an effect of all transformations (if any) attached to the attribute.

An example of a transfer's use will make it clear. You can use a transfer transformation to set the line width of one polygon equal to a half of the line width of another. Name the first polygon *PY1*, and the second *PY2*. Attach a transfer transformation to the *LineWidth* attribute of *PY1*. Constrain the *Source* attribute of the transformation to the *LineWidth* attribute of *PY2*. Now attach the *Divide* transformation to the *Buffer* attribute of the *Transfer* transformation, and set its *D* attribute to 2. The *LineWidth* of *PY1* is now constrained to be one half of the value of the *PY2 LineWidth* attribute.

Identity

An *Identity* transformation ignores the attribute's *Value* and sets the attribute's output value to the value of the transformation's *Source* attribute, which is its only attribute. The type of the *Source* attribute matches the type of the attribute the transformation is attached too.This transformation is used for the internal design of the GLG Control Widgets.

Predefined Dynamics

In addition to the stock transformation types listed above, predefined dynamics options are provided in the Builder for convenience. Predefined dynamics provide easy to use options for implementing the most common dynamic actions, for example *Blinking Alert*. The parameters of predefined dynamics are presented to the user as a simple list of public properties that define the its dynamic behavior.

Predefined dynamics is implemented using custom transformations, which represent a collection of several transformations wired together to implement a specific dynamic behavior, and present it to a user as a simple list of public properties that define its parameters. Custom transformations may be used by the system integrators to extend GLG editors with elaborate application-specific dynamics. The *Export Property* feature of the OEM version of the Graphics Builder is used to define custom transformations.

The dynamics are attached to the object attributes and edited the same way as any other dynamic transformation. Predefined dynamics available in the Builder are listed below.

Color List

The *Color List* dynamics use an integer index to select a color from a list. It works the same way as the *List* transformation and differs only in the way it lists its properties. It has the following properties:

ColorN
Colors to be used, where N is a zero-based color index.

ColorIndex
The zero-based index controlling which color to use.

Color Threshold

The *Color Threshold* dynamics compare an input scalar value with the list of thresholds and selects a corresponding color from a list of colors. It works the same way as the *Threshold* transformation and differs only in the way it lists its properties. It has the following properties:

ColorN
Colors to be used, where N is a zero-based color index.

ThresholdN
Thresholds to be used, where N is a zero-based threshold index. Threshold values must be specified in the increasing order.

InputValue
This value is compared with the thresholds and controls which color to use. The color corresponding to the first threshold smaller than the input value is used.

Color Blinking

The *Color Blinking* dynamics alternate an attribute's color between the two specified colors. It has the following properties:

Enabled
Enables or disables blinking. When set to 0, the blinking is disabled and the *OffColor* is displayed.

Interval
The blinking interval in seconds.

OnColor
The first color.

OffColor
The second color.

Color Alert

The *Color Alert* dynamics change an attribute's color when the monitored value goes out of the specified range. It has the following properties:

ActivateOnEqual
If set to 1 (*True*), the color is changed when the input value is equal to or exceeds the specified range. If set to 0 (*False*), the color is changed only when the value exceeds the range.

InputValue
The monitored value.

Interval
The blinking interval in seconds.

OnColor
The first color.

OffColor
The second color.

RangeHigh
The *High* range.

RangeLow
The *Low* range.

Color Blinking Alert

The *Color Blinking Alert* dynamics alternate the attribute's color between the default and alarm colors when the monitored value goes out of the specified range. It has the following properties:

ActivateOnEqual
If set to 1 (*True*), blinking starts when the input value is equal to or exceeds the specified range. If set to 0 (*False*), blinking starts only when the value exceeds the range.

ColorOK
The default color to use when the value is inside the *Low / High* range.

ColorWarning
The color to use when the value goes outside of the *Low / High* range.

ColorAlarm
The color to use when the value goes outside of the *LowLow / HighHigh* range.

InputValue
The monitored value.

Interval
The blinking interval in seconds.

RangeHigh
The *High* range.

RangeHighHigh
The *HighHigh* range.

RangeLow
The *Low* range.

RangeLowLow
The *LowLow* range.

List

The *List* dynamics use an integer index to select an attribute value from a list. It works the same way as the *List* transformation and differs only in the way it lists its properties. It may be applied to attributes of either D (double) or S (string) type and has the following properties:

ListIndex
The zero-based index controlling which attribute value to use.

ValueN or **TextStringN**
Attribute values or text strings to be used, where N is a zero-based value or string index.

Threshold

The *Threshold* dynamics compare an input scalar value with the list of thresholds and selects a corresponding attribute value from a list of values. It works the same way as the *Threshold* transformation and differs only in the way it lists its properties. It may be applied to attributes of either D (double) or S (string) type and has the following properties:

InputValue
This value is compared with the thresholds and controls which attribute value to use. The value corresponding to the first threshold smaller than the input value is used.

ThresholdN
Thresholds to be used, where N is a zero-based threshold index.The thresholds must be specified in the order of increasing their values.

ValueN or **TextStringN**
Attribute values or text strings to be used, where N is a zero-based value or string index.

Blinking

The *Blinking* dynamics alternate an attribute's value between the two specified values. It may be attached to any attribute of D type (double), including the object's visibility. It has the following properties:

Enabled
Enables or disabled blinking. When set to 0, the blinking is disabled and the *OffValue* is displayed.

Interval
The blinking interval in seconds.

OnValue
The first color.

OffValue
The second color.

Range Alert

The *Range Alert* dynamics changes the attribute's value when the monitored value goes out of the specified range. It has the following properties:

ActivateOnEqual
If set to 1 (*True*), the attribute's value is changed when the input value is equal to or exceeds the specified range. If set to 0 (*False*), the value is changed only when the input value exceeds the range.

InputValue
The monitored value.

ValueOK
The value to use when the value is inside the *Low / High* range.

ValueWarning
The value to use when the value goes outside of the *Low / High* range.

ValueAlarm
The value to use when the value goes outside of the *LowLow / HighHigh* range.

RangeHighHigh
The *HighHigh* range.

RangeHigh
The *High* range.

RangeLow
The *Low* range.

RangeLowLow
The *LowLow* range.

The *ValueWarning*, *RangeHighHigh* and *RangeLowLow* properties are not present in the *Range Alert* dynamics attached to the *Visibility* attribute, since visibility has only two states: ON and OFF.

Blinking Alert

The *Blinking Alert* dynamics alternate the attribute's value when the monitored value goes out of the specified range. It has the following properties:

ActivateOnEqual
If set to 1 (*True*), blinking starts when the input value is equal to or exceeds the specified range. If set to 0 (*False*), blinking starts only when the value exceeds the range.

InputValue
The monitored value.

Interval
The blinking interval in seconds.

OnValue
The value to use for blinking when the value goes outside of the range.

OffValue
The default value to use when the value is inside the range.

RangeHigh
The *High* range.

RangeLow
The *Low* range.

VisibilityThreshold

The *Visibility Threshold* dynamics compare an input scalar value with the specified threshold and sets the object visibility to a matching value. It has the following properties:

InputValue
This value is compared with the threshold and controls which attribute value to use.

Threshold
The threshold.

VisState0
The visibility value to use when the input value is less than the threshold. May be set to 0 or 1.

VisState0
The visibility value to use when the input value is less than the threshold. May be set to 0 or 1.

Value Display

The *Value Display* dynamics may be attached to the *TextString* attribute of text objects to display a numerical value using the specified format. It has the following properties:

InputValue
The value to be displayed.

Label
The label used to annotate the value.

MinLength
The minimum number of characters used to display the value. If the number of characters is less than *MinLength*, the value is padded with spaces on the left.

Precision
The number of digits after the decimal point in the value display.

Separator
The string used as a separator between the label and the value.

Units
The unit string displayed after the value.

Text Display

The *Text Display* dynamics may be attached to the *TextString* attribute of text objects to display a string using the specified format. It has the following properties:

InputString
The string to be displayed.

Label
The label used to annotate the string value.

MinLength
The minimum number of characters used to display the string. If the number of characters is less than *MinLength*, the string is padded with spaces on the left.

Separator
The string used as a separator between the label and the string.

Suffix
The second annotation displayed after the string.

Time Display

The *Time Display* dynamics may be attached to the *TextString* attribute of text objects to display the current time. It has the following properties:

Enabled
Enables or disables time updates. In the Builder, use the Run mode to see updates of the time display.

TimeFormat
A character string specifying a desired time format (*"%X "* by default). Refer to the description of the axis object's *Time Format* attribute on page 126 for information on the supported time formats.

TimeLabel
A label appended to the displayed time string.

UTCFlag
If set to 1, the UTC time will be displayed. Otherwise, a local time will be shown.

UpdateInterval
Update interval in seconds (1 by default).

Date Display

The *Date Display* dynamics may be attached to the *TextString* attribute of text objects to display the current date. It has the following properties:

Enabled
Enables or disables time updates. In the Builder, use the Run mode to see updates of the time display.

DateFormat
A character string specifying a desired date format (*"%x "* by default). Refer to the description of the axis object's *Time Format* attribute on page 126 for information on the supported time formats.

DateLabel
A label appended to the displayed time string.

UpdateInterval
Update interval in seconds (1 by default).

Flow

The *Flow* dynamics may be attached to the *LineType* attribute of lines and polygons to visualize flow of gases and liquids through pipes. The dynamics shifts a line type pattern along the length of the line to animate the flow. It has the following properties:

DisabledLineType
The line type used to visualize the pipe when the flow is disabled. The default value is 0 (solid line).

EnabledLineType
The line type pattern used for animation when the flow is enabled.

FlowEnabled
Enables of disables the flow animation. When set to 0, the flow is disabled.

FlowInterval
Controls the flow speed by defining a timer interval (in milliseconds) between the flow updates. The default value is 0.1.

FlowInversed
Inverses the flow direction if set to 1. The default value is 0 (direct flow).

Alarm Object

The alarm object can be attached to a data or attribute object to monitor its value. The alarm object defines the normal range of the monitored value and generates an alarm message when the value goes outside of the normal range. There is also a type of alarm that generates a message every time the monitored value changes.

Internally, the alarm object is implemented as a special type of a transformation object and its attributes follow the same convention for the default attribute names.

While alarm attributes depend on the type of alarm as described in the following sections, all alarm objects share the following common attributes:

Alarm Label
Contains a user-defined label used to identify the alarm.

Enabled
Used to enable or disable the alarm by setting its value to 1 or 0, respectively.

The rest of the alarm attributes depend on the alarm type and are described in the following sections.

Alarm Messages

Alarm objects generate alarm messages when changes of the monitored value trigger a specified alarm condition. Alarm messages are processed by an alarm handler which is installed by using the *GlgSetAlarmHandler* API method described on page 78 of the *GLG Programming Reference*

Manual. Each message contains *Action* and *SubAction* parameters that indicate the condition that generated the message. The message also contains *AlarmLabel*, as well as the alarm object and the attribute object whose value change triggered the alarm.

There are two types of alarm messages: messages that indicate the state of readiness of the alarm object, and messages that reflect the alarm state of the monitored value. The following alarm messages are generated by all alarm objects to reflect readiness of the alarm object:

- A message with the *Arm* action is generated when the drawing containing an alarm is drawn (setup).
- A message with the *Enabled* action is generated every time the alarm is enabled.
- A message with the *Disabled* action is generated every time the alarm is disabled.
- A message with the *Disarm* action is generated when the drawing containing the alarm is erased (reset).

When an alarm is armed and enabled, it generates alarm messages when changes of the monitored attribute value trigger the alarm condition. These messages vary depending on the alarm type and are described in the following sections.

Range Alarm

The *Range* alarm can be attached to an attribute of a D type (double) to monitor its value. The alarm message is generated when the value goes below or above the specified *High/Low* range.

The range alarm has the following attributes:

Use Value
If set to *Value* (default), the value of the attribute before applying any attribute transformations is used. If set to *XfValue*, the transformed value of the attribute is used. The transformed value includes an effect of all transformations (if any) attached to the attribute.

High
Defines the high range of the value. The alarm message with the *High* subaction is generated when the value reaches or exceeds the *High* range.

Low
Defines the low range of the value. The alarm message with the *Low* subaction is generated when the value drops below or equal to the *Low* range.

The *Range* alarm generates the following alarm messages:

- A message with the *Set* action is generated when the monitored value changes from being within the alarm's range to being out of range. The *SubAction* parameter of the message indicates the alarm condition, *High* or *Low.*
- A message with the *On* action is generated when the monitored value outside of the range changes to a value which is still outside the range, with the same alarm state (*High* or *Low*) as before the change.The *SubAction* parameter of the message indicates the alarm state, *High* or *Low.*
- A message with the *Reset* action is generated when the monitored value changes from being outside of the range to within the range. The *SubAction* parameter of the message indicates the alarm state being reset, *High* or *Low.*

- If the alarm state changes from one alarm state to another, for example from *High* to *Low*, a message with the *Reset* action is generated for the previous alarm state, then a message with the *Set* action is generated for the new alarm state.

Range2 Alarm

This alarm is similar to the *Range* alarm, but adds the following attributes that define second thresholds in addition to *High* and *Low*:

High High
Defines the *high high* range of the value. The alarm message with the *HighHigh* subaction is generated when the value reaches or exceeds the *HighHigh* range.

Low Low
Defines the *low low* range of the value. The alarm message with the *LowLow* subaction is generated when the value drops below or equal to the *LowLow* range.

The alarm generates the same messages as the *Range* alarm, with the addition of the alarm messages with the *HighHigh* and *LowLow* subactions for the corresponding alarm states.

Change Alarm

The *Change* alarm can be attached to an attribute of any type (D, S or G) to monitor changes of its value. The change alarm does not define any new attributes in addition to *Alarm Label* and *Enabled*.

The *Change* alarm generates an alarm message with the *ValueChange* action and NULL subaction every time the value is changed.

Action Object

An action can be attached to a graphical object to perform a specified action when user interacts with the object at run-time. There are three types of actions that differ based on the activation conditions:

- *Mouse action* is activated when a user clicks on the object with the mouse or moves the mouse over the object. Mouse actions may be attached to any graphical object in the drawing.
- *Input action* may be attached to an input object, such as a button or a slider, to perform a specified operation when the user interacts with the input object. Input actions may be attached only to input objects, and their activation conditions specify the exact input activity that triggers the action.
- *Tooltip action* is a special type of action that is used to define an object tooltip.

The type of an action is determined by its *Trigger* attribute which is set at the time the action is attached to an object. The trigger attribute may be set to either MOUSE_CLICK or MOUSE_OVER for mouse actions, and is always set to INPUT for input actions, and TOOLTIP for tooltip actions. A viewport's ProcessMouse attribute must be set to a value that includes a combination of *Click*, *Move* and *Tooltip* masks to enable processing corresponding mouse actions.

There are also several types of action objects based on the type of activity performed when the action is triggered:

- *Command action*

  Command actions define an operation to be performed when the action is triggered and contain additional data needed to execute it. For example, a *GoTo* command may be attached to a button to navigate to another graphical page when a user clicks on the button. The command will include a parameter that specifies a filename of the page to be loaded.

  When a command action is attached to an object in the Builder, a list of commands is displayed in a dialog; the selected command will be associated with the action. Each command has a *CommandType* parameter, as well as a number of other parameters depending on the command type.

  By default, the Builder uses a predefined list of commands; new custom command types can be added by customizing the Builder as described in the *Custom Data Sets and Custom Commands* section on page 287. The available command types are listed in the *Command Object* section on page 184.

  At run-time, the input callback will be invoked with *Format=Command* when the command is triggered. Refer to the *Handling Action Object Messages and Commands in Application Code at Run Time* section on page 188 for information on using command actions in the application code at run time.

- *Custom event action*

  These actions generate custom events and may be used by an application to trace specific types of events. For example, a custom event may be generated every time a mouse is moved over an object, or an object is selected with the mouse click. Unlike the command actions, a custom event is generated on both action activation as well as reset (the mouse moves away from the object or a mouse button is released).

  Custom event actions are also backward compatible with the custom event handling code from previous releases of the Toolkit (prior to v. 3.5), while providing a better encapsulation and more precise activation conditions.

  Custom event actions may contain additional custom data as part of the action object. At run-time, the input callback will be invoked with *Format="CustomEvent"* when the custom event is triggered. Refer to the *Handling Action Object Messages and Commands in Application Code at Run Time* section on page 188 for information on using custom event actions in the application code at run time.

- *Mouse feedback action*

  Mouse feedback actions are used to change appearance of an object without writing any application code. For example, it may be used to highlight an object on mouse click or mouse over by changing its *LineWidth* or *FillColor* attribute. Mouse feedback actions have a *State* parameter; the value of the parameter is set by the action depending on the requested mouse activation condition.

  To change an object's appearance, the *State* parameter of the action should be constrained to some attribute of the object. For example, to change the object's color on mouse over, *State* can be constrained to the index of a color list transformation attached to the object's

FillColor attribute.

There are several types of available feedback:

TRACE_STATE
: Traces the state of the mouse click on the object or the mouse being over the object (as requested by the action's *Trigger* parameter) by setting the action's *State* parameter to 1 or 0. It can also trace the state of the *Control* key if requested by the action's *ProcessArmed* parameter, in which case the value of *State* will be set to 2 if the mouse activation condition specified by *Trigger* are met and the *Control* key is pressed. See page 180 for more information.

SET STATE
: Sets the action's *State* attribute to 1 when the action's activation conditions are met.

RESET_STATE
: Resets the action's *State* attribute to 0 when the action's activation conditions are met.

TOGGLE_STATE
: Toggle the action's State attribute every time the action is activated.

The TRACE_STATE feedback type is usually used to provide a visual feedback when the object is selected with a mouse click or a mouse over event. The SET_STATE and RESET_STATE feedback types allow the application to use a graphical object as a button that sets or resets the state of some other object in the drawing, without the use of button input objects, while the TOGGLE_STATE allows the application to use an object as a toggle button.

Advanced: Mouse feedback actions automatically update the viewport that contains the object to which the mouse feedback action is attached. If the action's *State* attribute is constrained to an object in a different viewport (that is not a child of the viewport containing the object with the action), that viewport will not be updated. The feedback actions generates an *UpdateDrawing*, which can be processed in the input callback to update the top-level or sibling viewport that contains the constrained object, if required. Refer to the *Handling Action Object Messages and Commands in Application Code at Run Time* section on page 188 for more information.

- *Tooltip action*
  Activates a tooltip when the mouse hovers over an object.

Advanced: At run time, the drawing traces the mouse position and the state of mouse buttons, processing appropriate actions for all objects selected with the mouse. If an object has several actions that match the mouse event, all of them will be executed: the mouse feedback actions will be processed first, then the custom event actions, and the command actions will be processed last. For TRACE_STATE, SEND_EVENT and SEND_COMMAND actions, if several actions of the same type are attached to an object, only the first action of each type will be executed. For the SET_STATE and TRACE_STATE actions, all actions attached to an object will be executed.

In cases when several intersecting objects are potentially selected, the objects drawn on top are processed first. If both a group and its elements have actions attached, the objects at the bottom of the object hierarchy (group elements) are processed first, before processing the group's actions.

For each combination of an action type (custom event action, command action or mouse feedback action) and *Trigger* type (mouse click or mouse over), action processing stops after finding the first object with matching actions and executing these actions. If an object has several actions of the same type attached, all of them will be processed.

For other *ActionType/Trigger* combinations that have not yet been handled, the search for matching actions continues until it processes all selected objects.

When processing actions, the old-style (prior to v. 3.5) custom events and mouse feedback properties of the drawing are handled the same way as actions: the action processing stops when the first property matching the event has been processed. For example, is an object has the *MouseClickEvent* property, a custom event will be generated and the search for the mouse click event actions will stop.

The action object provides a more efficient alternative to the old-style custom events, mouse feedback and tooltip properties (such as *MouseClickEvent*, *TooltipString*, etc.) since it does not involve resource name queries required by these properties. If the drawing does not contain the old-style properties, the *GlgDisablePre35ObjectEvents* global configuration resource or the GLG_DISABLE_PRE35_OBJECT_EVENTS environment variable may be set to speed up mouse move processing for large drawings.

Action Object Attributes

Most of the action attributes are common across all available action types, with the exception of the tooltip action that has only two attributes: *Tooltip* and *Enabled*. The following lists all attributes of an action object:

ActionType

Defines the type of the action performed when the action is triggered, may have one of the following values:

SEND_EVENT

Generates a custom event with the event label defined by the action's *EventLabel* attribute. The custom event provides backward compatibility with the custom event handling code from previous releases of the Toolkit. Additional custom data needed by an application may be held in the *ActionData* container, which can be accessed via the *Add Data* or *Edit Data* button in the Builder. The action also allows the user to define an arbitrary set of action data, which is different from the SEND_COMMAND action that uses predefined sets of data for each command type.

If *ProcessArmed=ARMED*, the custom event will be sent only if the *Control* key was held down. If *ProcessArmed=ARMED and UNARMED*, the event will be sent regardless of the state of the *Control* key, but the *SubAction* resource of the message object in the input callback will be set to *"Armed"* if the *Control* key was pressed.

Refer to the *Handling Action Object Messages and Commands in Application Code at Run Time* section on page 188 for information on using command actions in the application code at run time.

SEND_COMMAND

Generates a command event with a *Command* object containing command type, as well as data required to execute the command. The command data are held in the *Command* container, which is accessible via the *Edit Command* button in the Builder.

When a command action is created, the Builder prompts the user to select a command from a list of several predefined command types, and then displays the Properties dialog for the selected command. Refer to the *Command Object* section on page 184 for the list of the predefined command types. The set of predefined command types may be extended by adding custom commands to the list, see the *Custom Data Sets and Custom Commands* section on page 287 for more information.

If *ProcessArmed=ARMED*, the command will be sent only if the *Control* key was held

down. If *ProcessArmed=ARMED and UNARMED*, the command will be sent regardless of the state of the *Control* key, but the *SubAction* resource of the message object will be set to *"Armed"* if the *Control* key was pressed.

Refer to the *Handling Action Object Messages and Commands in Application Code at Run Time* section on page 188 for information on using command actions in the application code at run time.

TRACE_STATE

Traces the mouse events and sets the action's *State* attribute depending on the action's *Trigger*:

- *Trigger = MOUSE_CLICK*

  If *ProcessArmed=NONE,* sets *State* to 1 when the object is clicked with the mouse button specified with the *MouseButton* attribute, and resets *State* back to 0 when the mouse button is released.

  If *ProcessArmed=ARMED*, the value of *State* is changed to 1 only if the *Control* key is pressed down when the mouse click occurs, and is reset back to 0 if the either the *Control* key or the mouse button is released.

  If *ProcessArmed=ARMED and UNARMED*, the *State* changes to 1 on the mouse click without the *Control* key, or to 2 if the *Control* key was held down. The *State* returns back to 1 when the *Control* key is released, and to 0 when the mouse button is released.

- *Trigger = MOUSE_OVER*

  If *ProcessArmed=NONE,* sets *State* to 1 when the mouse moves over the object and resets *State* back to 0 when the mouse moves away from the object.

  If *ProcessArmed=ARMED*, the value of *State* is changed to 1 only if the *Control* key is pressed down and the mouse is positioned on top of the object.

  If *ProcessArmed=ARMED and UNARMED*, the *State* changes to 1 when the mouse moves over the object without the *Control* key, or to 2 if the *Control* key is held down while the mouse is positioned on top of the object. The *State* returns back to 1 when the *Control* key is released, and to 0 when the mouse moves away from the object.

SET_STATE

Sets the action's *State* attribute to 1 when the action is triggered.
If *ProcessArmed=ARMED*, the action is activated only if the *Control* key is held down.

RESET_STATE

Resets the action's *State* attribute to 0when the action is triggered.
If *ProcessArmed=ARMED*, the action is activated only if the *Control* key is held down.

TOGGLE_STATE

Toggles the action's *State* attribute between 0 and 1 every time the action is triggered.
If *ProcessArmed=ARMED*, the action is activated only if the *Control* key is held down.
Only the MOUSE_CLICK triggers are supported for TOGGLE_STATE.

Trigger

Specifies an action's activation condition, may have one the following values:

MOUSE_CLICK

Activates the action when the object is selected with the mouse button specified by the *MouseButton* attribute. The *ProcessArmed* attribute may be used to handle the state of the *Control* key.

A viewport's *ProcessMouse* attribute must contain the *Click* mask for the action to be processed.

MOUSE_OVER

Activates the action when the mouse moves over the object. The *ProcessArmed* attribute may be used to handle the state of the *Control* key.

A viewport's *ProcessMouse* attribute must contain the *Move* mask for the action to be processed.

INPUT (for input actions only)

Activates an action attached to an input object when activity specified by the *InputAction* is detected. For example, *InputAction=ValueChanged* may be used to activate the action attached to a slider every time the slider value is changed.

The value of the *InputAction* attribute is a string that matches the *Action* resource of a message object received in the input callback. The action's *InputSubAction* attribute may be used to match the *SubAction* attribute of the message object is required.

Input actions may be used to effectively translate user interaction with input objects into encapsulated action commands that may also contain additional data needed for executing the command. For example, a *GoTo* command contains a *DrawingFile* parameter that specifies the drawing to navigate to.

The use of input actions makes it possible to attach well-defined commands to input objects, instead of using input object names for handling user interaction. In the previous releases of the Toolkit (prior to v. 3.5), an application code in the input callback had to use input object names and the *Action* / *SubAction* parameters of the message object to determine the action to be performed. Using input actions, the code can handle a set of predefined commands that are completely defined in the drawing, making the code more generic and independent of object names.

Refer to the *Handling Action Object Messages and Commands in Application Code at Run Time* section on page 188 for information on using input actions in the application code at run time.

TOOLTIP (for tooltip actions only)

Displays a tooltip when the mouse is hovering over an object. The value of the attribute is set at the creation time when the tooltip action is attached to an object, and the *Trigger* attribute is not shown in the tooltip action's properties.

ProcessArmed (mouse actions only)
Modifies an action's activation condition to check the state of the *Control* key. In mission-critical applications, the *Control* key is always used to "arm" the action, so that it could be activated ("armed") only if the *Control* key is held down and inactive ("disarmed") if the *Control* key is not pressed. The attribute may have one the following values:

NONE
An action may be activated regardless of the state of the *Control* key.

ARMED
An action may be activated only if the *Control* key is held down.

ARMED & UNARMED
Modifies the behavior of the TRACE_STATE action to set its *State* attribute to different values depending on the state of the *Control* key. For other action types, this setting is the same as NONE.

MouseButton (mouse click actions only)
Specifies the index of the mouse button (0, 1 or 2) that will trigger the action for actions with the MOUSE_CLICK trigger.

Enabled
Enables or disables the action when is set to 1 or 0.

EventLabel (SEND EVENT and SEND COMMAND actions only)
Defines an event label string that will be available in the input callback as the *EventLabel* resource of the message object. The event label may be used to differentiate between different custom events or action commands in the input callback.

State (SET STATE, RESET STATE and TOGGLE STATE actions only)
The output parameter whose value will be set to reflect the mouse events based on the action type. Refer to the description of the *ActionType* attribute on page 179 for information on possible values of the *State* attribute for different action types.

The *State* is an attribute of D (double) type and may be constrained to some D attribute of another object in the drawing to provide a visual feedback. For example, to provide a *MouseOver* feedback via changing the object's color, *State* can be constrained to the index of a color list transformation attached to the object's *FillColor* attribute. The *State* attribute can also be constrained to an attribute of a different object, to change that object's appearance when the object the action is attached to is selected with the mouse.

InputAction (input actions only)
Specifies when the Input Action attached to the input object is triggered. For example, *InputAction=ValueChanged* may be used to activate the action attached to a slider every time the slider value is changed, and *InputAction=Activate* may be used to trigger an action attached to a push button.

When an input object receives user input, a default message containing the *Action* and *SubAction* resources describing the type of the input activity is generated.

At run time, the default message can be processed by the application code in the input callback to perform various operations depending on the name of the input object, as well as the *Action* and *SubAction* resources supplied by the message. This technique of handling user input in the application code does not require any

input actions to be added to input objects at design time. However, it makes the application code dependent on the names assigned to input objects in the drawing. To make the code more generic and independent of object names, Input Actions may be attached to input objects at design time in the GLG Builder (Enterprise Edition) or HMI Configurator.

When an Input Action is attached to an input object, the action object receives the default message generated by the input object and checks its *Action* and *SubAction*: if they match the *InputAction* and *InputSubAction* of the action object, the action is activated, sending an action message to the input callback for SEND_EVENT or SEND_COMMAND actions.

Instead of processing the default message generated by an input object, the input callback code can process the message generated by the Input Action attached to the input object. This makes it possible to define various commands in the drawing at design time and to free the application code in the input callback from a dependency on hardcoded input object names. An action may also contain additional data needed to execute the command, which makes it easier to assign predefined commands to objects in the drawing, especially for the users of the HMI Configurator.

At run time, the *ActionObject* resource of the message received in the input callback will contain the action object that generated the message. For command actions, the command may be accessed via the *ActionObject/Command* resource. The command type may be accessed as the *CommandType* resource of the *Command* object (for the Intermediate API), or directly from the message via the *ActionObject/Command/CommandType* resource (for the Standard API). The rest of the command data can be accessed via the corresponding resource names.

Both the default input object message and the action message are passed to the input callback, and the application can decide to handle one or another, or both.

InputSubAction (input actions only)
Specifies an optional input object's subaction that triggers execution of the Input Action. If InputSubAction is defined, the action is activated if both *InputAction* and *InputSubAction* match the corresponding resources of the input object's message. If *InputSubAction* is not defined, only *InputAction* is checked.

The *InputSubAction* attribute is not shown in the action properties, but can be accessed as an action's resource via the Resource Browser in the Builder or via the GLG API at run-time. By default, the value of the attribute is set to an empty string.

Add Data / Edit Data (SENT_EVENT actions only)
A button in the Action Properties dialog for accessing *ActionData* of the SEND_EVENT action. The *ActionData* container holds any action data that may be needed for processing the action. When action data are added, the Builder displays a choice of a predefined custom data set or adding data elements one by one manually.

The predefined custom data set contains the *DataSetType* parameter set to "Custom" as well as several string and numeral parameters named *ParamS*, *ParamS1*, *ParamS2* and *ParamD*, *ParamD1*, *ParamD2*. Predefined custom data sets are edited the same way as the command data described below. Additional custom data sets may be defined by customizing the Builder as described in the *Custom Data Sets and Custom Commands*

section on page 287.

If the action data are to be defined manually, a list-based interface for defining individual action parameters is displayed. This interface is the same as the interface used to define custom data attached to an object, see the *Edit Custom Properties* section on page 350.

Refer to the *Handling Action Object Messages and Commands in Application Code at Run Time* section on page 188 for information on accessing action data in the application code at run time.

GLG API note: *ActionData* is a group object that contains action data. If a custom data set was selected, it contains public properties that represent action data.

Add Command / Edit Command (SEND_COMMAND actions only)
A button in the Action Properties dialog for accessing *Command* of the SEND_COMMAND action. The *Command* object contains the *CommandType* attribute as well as other data used to execute the command.

The command associated with a command action is selected from a predefined list of available commands at the time the action is attached to an object. New custom command types can be added by customizing the Builder as described in the *Custom Data Sets and Custom Commands* section on page 287. The available command types and their parameters are listed in the *Command Object* section on page 184.

By default, the action's command data are displayed for editing as properties in the Action Properties dialog. The *Options, Selection Options, Edit Action Data as List* menu option may be used to edit the properties as a list, which allows deleting or adding new data to the command. The list editing interface is the same as the interface for editing custom properties, see the *Edit Custom Properties* section on page 350. A button  in the upper right corner of the dialog provides a convenient shortcut for switching command data display.

To delete a command, delete the action containing the command. To delete a command without deleting the action object, edit the command data as a list and delete all command properties. A new command can then be added via the Add Command button in the Action Properties dialog.

Refer to the *Handling Action Object Messages and Commands in Application Code at Run Time* section below for information on accessing command data in the application code at run time.

GLG API note: a command is a group object that contains public properties that define command data.

Command Object

A command object is a group that contains all command data. A default set of predefined commands is provided; new custom command types can be added by customizing the Builder as described in the *Custom Data Sets and Custom Commands* section on page 287.

All commands contain a mandatory *CommandType* resource, which is a string that defines the type of an operation to perform. The rest of the parameters differ depending on the command type. The following lists all available command types, the additional data each command contains, as well as the suggested use of the data in an application code. The suggested use is just an example of how the data could be used; an application can decide to use the data in a different way if needed. Each data element of a command includes its name and data type (S for strings or D for double values).

GoTo

CommandType (S)
Command type, is set to "GoTo". The command is used to navigate to a different drawing. For example, a drawing can contain *Next* and *Previous* buttons that are used to switch the the currently displayed drawing.

DrawingFile (S)
The filename of the drawing to load.

Destination (S)
The name or the resource name of the container used to display the drawing. If a top-level drawing contains a subwindow used to display different drawings, *Destination* points to that subwindow. It may be left empty to replace the top-level drawing with a new drawing or to use a default destination.

Title (S)
Specifies an optional title string to be used by an application as needed.

ParamS (S)
An optional string parameter.

ParamD (D)
An optional numerical parameter.

PopupDialog

CommandType (S)
Command type, is set to "PopupDialog". The command displays a popup dialog.

DialogType (S)
Specifies an optional dialog type in case an application uses several popup dialogs (such as drill-down dialog, alarm dialog, set value dialog, etc.).

DialogResource (S)
The name or the resource path of the dialog object to be shown.

DrawingFile (S)
The filename of a drawing to load into the dialog in case the dialog contains a subwindow that displays different dialog drawings depending on the context. May be left empty if a viewport with a fixed drawing is used as a dialog object.

Destination (S)
The name or the resource path of the subwindow object to load the dialog drawing into. It may be different from *DialogResource* in cases when a viewport (*DialogResource*) is used as a dialog object, and a subwindow (*Destination*) inside the viewport is used to load different context-dependent dialog drawings.

Title (S)
Specifies an optional title string to be used by an application as needed.

ParamS (S)
An optional string parameter.

ParamD (D)
An optional numerical parameter.

PopupMenu

CommandType (S)
Command type, is set to "PopupMenu". The command displays a popup menu.

MenuType (S)
Specifies a popup menu type in case an application uses several popup menus.

MenuResource (S)
The name or the resource path of the popup menu object to make visible.

DrawingFile (S)
The filename of a drawing to load into the popup menu in case the popup menu contains a subwindow that displays different menu drawings depending on the context. May be left empty if a viewport with a fixed drawing is used as a popup menu.

Destination (S)
The name or the resource path of the subwindow object to load the popup menu drawing into. It may be different from *MenuResource* in cases when a viewport (*MenuResource*) is used as a popup menu, and a subwindow (*Destination*) inside the viewport is used to load different context-dependent menu drawings.

Title (S)
Specifies an optional title string to be used by an application as needed.

ParamS (S)
An optional string parameter.

ParamD (D)
An optional numerical parameter.

ClosePopupDialog

CommandType (S)
Command type, set to "ClosePopupDialog", is used to close a popup dialog.

DialogType (S)
Specifies a type of a previously displayed popup dialog to be closed in cases when an application uses several popup dialogs.

ParamS (S)
An optional string parameter.

ParamD (D)
An optional numerical parameter.

ClosePopupMenu

CommandType (S)
Command type, set to "ClosePopupMenu", is used to close a popup menu.

MenuType (S)
Specifies a type of a previously displayed popup menu to be closed in cases when an application uses several popup menus.

ParamS (S)
An optional string parameter.

ParamD (D)
An optional numerical parameter.

WriteValue

CommandType (S)
Command type, set to "WriteValue". This command is used to write a value into the process database. For example, a *Start Motor* button may be used to write a value of 1 into the tag of the process controller that controls the motor, and a *Stop Motor* button will write the value of 0.

OutputTagHolder (D)
This attribute is a placeholder for an attached output tag. The *TagSource* attribute of the tag specifies the tag field of the process controller to write the value to. A data browser (if provided) may be used to select the tag source from a list of available tags.

Value (D)
The value to be written to the tag specified by *OutputTagHolder*.

ParamS (S)
An optional string parameter.

ParamD (D)
An optional numerical parameter.

WriteValueFromWidget

CommandType (S)
Command type, set to "WriteValueFromWidget". This command is used to write the current value of the input widget (such as a numerical text input box, a spinner or a slider) into the process database.

OutputTagHolder (D)
This attribute is a placeholder for an attached output tag. The *TagSource* attribute of the tag specifies the tag field of the process controller to write the value to. A data browser (if provided) may be used to select the tag source from a list of available tags.

ValueResource (S)
The resource name of the resource inside the input widget that contains the value. For example, "Value" may be used as a resource name for accessing the value of a GLG spinner.

ParamS (S)
An optional string parameter.

ParamD (D)
An optional numerical parameter.

Custom

CommandType (S)
Command type, set to "Custom". This command may be used to define additional custom commands by changing its *CommandType* attribute.

New custom commands with command-specific data may be added by customizing the Builder or HMI Configurator. Refer to the *Custom Data Sets and Custom Commands* section on page 287 for more information.

ParamS (S)
An optional string parameter.

ParamD (D)
An optional numerical parameter.

CustomExt

The *CommandExt* command type is the same as Command, but with the following additional parameters for storing command data:

ParamS2 (S)
ParamS3 (S)
Optional string parameters.
ParamD2 (D)
ParamD3 (D)
Optional numerical parameters.

Handling Action Object Messages and Commands in Application Code at Run Time

When a SEND_EVENT or SEND_COMMAND action attached to an object is activated, it generates a message which is received by an application's input callback as the *message* parameter. The message contains resources that identify the event that triggered the action.

Custom Event Message

This message with *Format=CustomEvent* is generated by the SEND_EVENT actions. The *Action* and *SubAction* resources of the message contain information about the input event, the *EventLabel* resource contains the *EventLabel* parameter of the action object, and the *ActionObject* contains the action that generated the message. The *ActionData* resource of the action object contains action data and may be accessed directly from the message object via as *ActionObject/ActionData*.

Refer to the *Custom Event Message Object* section on page 360 of the *GLG User's Manual and Builder Reference* for more information.

The message object of the SEND_EVENT action is the same as the *CustomEvent* message used in the previous releases of the Toolkit (prior to v. 3.5), with the only difference of the new *ActionObject* parameter.

Command Message

This message has *Format=Command* and is generated by the SEND_COMMAND actions. A different value of the *Format* parameter allows to easily differentiate between custom event and command messages. The *EventLabel* resource of the message object contains *EventLabel* of the action object, and the *ActionObject* resource contains the action that generated the message.

The command message is processed in the application code by using command type and other command data contained in the command object. The command object can be accessed as a *Command* resource of the action object, and may be accessed directly from the message object of the input callback via the *ActionObject/Command* resource.

The command object contains the command's data, such as *CommandType* and other resources, depending on command type. These resources may be accessed as resources of the command object, or directly from the message object. For example, to query a command type directly from the message object, the following resource may be used: *ActionObject/Command/CommandType*.

The rest of the command data can be accessed via their corresponding resource names. Refer to the *Command Message Object* section on page 363 of the *GLG User's Manual and Builder Reference* for more information.

UpdateDrawing Message

This message has *Format=UpdateDrawing* and is generated by the TRACE_STATE, SET_STATE and RESET_STATE mouse feedback actions, which may need to update the drawing after changing the value of their *State* attribute.

Mouse feedback actions automatically update the viewport that contains the object to which the mouse feedback action is attached. If the action's *State* attribute is constrained to an object in a different viewport (that is not a child of the viewport containing the object with the action), that viewport will not be updated. To handle this case, a mouse feedback action generates the *UpdateDrawing* message with message's *Action=Update*. The application may choose to process this message in the input callback to update the top-level or sibling viewport that contains the constrained object, if needed.

Refer to the *UpdateDrawing Message Object* section on page 365 of the *GLG User's Manual and Builder Reference* for more information.

Chapter 5

Input Objects

5

Widgets such as Sliders, Dials, Buttons and others that are capable of reacting to input events are called **input widgets**. These can allow a user to control a GLG drawing, either directly, through constrained drawing attributes, or indirectly, through a user application that handles input events. The ability to build custom input widgets is a necessary attribute for creating an open interactive graphical environment.

The GLG Toolkit offers two options for creating input widgets:

- You can use GLG graphical objects to render the widget.
- You can use the *WidgetType* attribute of a viewport's screen object to select one of the available **native widget** types (push button, scrollbar, etc.) to render the widget. "Native" refers to the windowing environment in which GLG is running: Windows, X Windows/Motif, Java or .NET. For example, a native button will appear as a Windows button if the drawing is displayed on Windows, as a Motif button if the drawing is X/Motif environment, or as a Swing button if the drawing is displayed in Java.

In both cases, the *Handler* attribute of a viewport object has to be used to specify a GLG **input handler**, which accepts user input, changing the widget's resources and visual appearance accordingly. For example, a slider widget reacts to mouse events by moving the graphical element that indicates its current position and updating the slider's *Value* resource. The handler also generates messages passed to the application's **input callback**, allowing the program to react to the input events.

Some input widgets, such as a toggle or slider, keep value or state information and change corresponding resources of the widget's drawing, making it possible to constrain other elements of the drawing to the resources of the input widget. For example, the *Visibility* attribute of an object in the drawing may be constrained to the *OnState* resource of the toggle button. The toggle's handler will change the value of the toggle's *OnState* resource each time it is clicked on with the mouse, changing the visibility of the constrained object in the drawing without a need to write any supporting code.

Each time the toggle changes its state, the toggle widget's handler also generates a *ValueChanged* message. At run time, the application's input callback is invoked to receive the message and provide additional application-specific handling of the event.

Other input widgets, such as a push button, don't keep any state information or resources that may be used to control other objects in the drawing. These widgets rely on the input callback to handle user interaction with the widget.

Input Handlers

A variety of input handlers are available. The handlers are attached to a viewport, where they look for a specific set of resources to control. For example, the *GlgKnob* handler used by the dial widgets controls a resource called *Value* (among others). When the handler is attached, the handler looks for a resource with that name. When the drawing is run, mouse movements in the viewport will be interpreted by the knob handler and translated into values of the *Value* resource.

The behavior of an input handler may be modified by defining certain resource names it recognizes. For example, the *GlgButton* handler searches for the resource named *OnState*. If it finds this resource, the handler implements a toggle, otherwise it implements a push button.

The resources controlled by an input handler must be visible at the top level of the widget viewport. *Alias* objects may be used to make resources defined inside the hierarchy to be visible at the top level. If widget resources are changed in the Builder, the widget's drawing has to be reset using the *File, Reset* menu option to allow the handler to update resource information.

The handler's resources appear not only in the widget viewport, but also in the message object passed by the handler to an input callback function. In addition, several of the input handlers define special resources that only appear in the message object. These resources are described with the description of each message object in the *Callback Events* section on page 97 of the *GLG Programming Reference Manual*.

The handlers available are as follows:

GlgSlider

Interprets linear mouse movement. Used for scrollbars, switches and sliders.

GlgNSlider

A native slider handler that takes advantage of local window system graphical representations.

GlgKnob

Interprets angular mouse movement. Used to implement dials, meters, knobs and switches.

GlgButton

Accepts mouse clicks for toggle and push buttons.

GlgNButton

A native button handler that works with native buttons, toggles and check box controls.

GlgNText

A native text handler that takes advantage of native text input widget. Works with both single and multi-line text edit controls.

GlgNList

A native list handler that handles native list widget in a cross-platform way. Works with both single and multiple selection list controls.

GlgNOption

A native option menu handler that works with the native option menu and combo box controls.

GlgMenu

Assembles buttons into a menu or a radio box.

GlgBrowser

A specialized browser for selecting object resources, tags and alarms.

GlgFontBrowser

A specialized browser, optimized for browsing X Windows fonts.

GlgClock

Displays the time. Can also be used as a stopwatch to record elapsed time.

GlgTimer

Triggers periodic updates with a specified rate. May be used to attach various blinking action to objects. It is superseded by more flexible *Timer* transformation.

GlgPalette

A specialized menu allowing a user to select arbitrary objects.

Attaching an Input Handler

To make the widget sensitive to the input events, an **input handler** is attached to the widget internally. The input handler is a block of code that reacts to the incoming events, changes the widget's appearance and calls the input callback when some event is translated into a change in the widget's state. For example, a slider widget reacts to mouse events by moving the graphical element that indicates its current position and changing the slider's *Value* resource. Depending on the handler type, an input handler recognizes a certain set of resources that control the handler's behavior.

An input handler is attached to a viewport with the viewport's *Handler* attribute. This attribute is a character string identifying which of the available handlers is to be used. To use the handler, you must also set the viewport's *DisableInput* attribute to NO (default).

To use a handler, you must equip the viewport with the resources the handler needs to operate. A knob widget, for example, must have a *Value* resource controlling some aspect of the drawing in such a way that changing the resource value in the range from 0 to 1 rotates the knob from the preferred minimum to maximum angle. You may also define optional resources that provide additional information to the handler, such as *Increment* or *Granularity.* These resources may be added to the widget's viewport as custom properties, using the *Object, Custom Properties, Add Custom Property* menu in the Enterprise Edition of the Builder. In the Basic and Professional Editions, named resources in the drawing may be used.

The sections below describe each of the available handlers, and list the resources they control.

Examples of Creating Custom Input Widgets

While the GLG Control Widget Set provides a variety of ready to use input widgets, the following examples illustrate input handlers' use by creating basic toggle and slider widgets from scratch. Refer to the following section for detailed description of the resources used in the examples.

Here are the steps to create a simple toggle button:

1. Create a drawing with a viewport and an object in the viewport, such as a small rectangle, that will be the indicator of the toggle state. You can also place a text label next to the rectangle.

2. Bring the rectangle's properties dialog, click on the ellipsis button ... next to the *Visibility* attribute and name the attribute *"OnState"*.

3. Set the viewport *Handler* attribute to "*GlgButton*". Make sure that the *DisableInput* attribute is set to NO.

4. Prototype this drawing using the *Start* toolbar button and select *Skip Command* in the Animation Command dialog. The *GlgButton* handler will toggle the rectangle's *Visibility* (the *OnState* resource) every time the button is pressed. Use the *Stop* toolbar button to exit the Run mode.

Here are the steps to follow to create a simple horizontal slider:

1. Create a drawing with a viewport, and an object in the viewport that is to be the indicator of the slider motion. This can be any shape or group of shapes. We will call this object the active element.

2. Assign a move transformation to the active element. Name the *Factor* attribute of the transformation *"ValueX"* and make sure that this attribute is visible at the top level of the viewport. (Set the viewport *HasResources* flag to YES and the active element's *HasResources* flag to NO. Alternatively, an alias object named *"ValueX"* may be added to the viewport to specify the full path to the *ValueX* resource.) Edit the move distance and the initial position of the active element to make sure that the values of *Factor* in the range from 0 to 1 correspond to the active element moving from the left side of the viewport to the right.

3. Set the viewport *Handler* attribute to *"GlgSlider"*. Make sure that the *DisableInput* attribute is set to NO.

When you run this drawing, the *GlgSlider* handler will read input from the cursor position when you click in the viewport, and use that position to set the value of the viewport's *ValueX* resource.

Of course, the *ValueX* resource need not control the position of an object. A move transformation attached to the active element is what we expect to see, but the resource could be attached to anything. For example, you could rotate a joystick in three-dimensions based on the linear position of the mouse.

To make the slider granular, create a scalar data resource named "*Granularity*" at the top level of the slider viewport's resource hierarchy. The value of the object indicates the number of positions the slider may take.

In the Enterprise Edition of the Builder, you can use the *Object, Custom Properties, Add Custom Property* menu to add a D property named "*Granularity*" to the slider's viewport. In the Basic and Professional Editions, create a dummy marker object to hold the resource, name its *MarkerSize* attribute "*Granularity*" and set the marker's *HasResources* flag to NO to make the *Granularity* resource visible at the viewport's top level. When editing is finished, reset the drawing using the *Reset* toolbar button and run the drawing to check the slider's new behavior.

For more information on the internal design of the input widgets, see page 216.

Common Input Handler Resources

All input handlers support the *ActiveState* resource:

ActiveState

The value of this resource is set to 0 when the input handler is disabled by setting the viewport's *DisableInput* attribute to YES. The *ActiveState* attribute may be attached to some resource in the drawing to alter widget's appearance in the inactive state.

Resources with the Param suffix

The output resources of input handlers, such as *Value, ValueX, ValueY, OnState* and others, have a corresponding resource whose name is formed by adding the *"Param"* suffix: *ValueParam, ValueXParam, OnStateParam* and so on. These resources are used only in the internal design of the input widgets and can be ignored in the application code.

For information on the internal design of the input widgets, see page 216.

GlgSlider

A slider is used to convert a linear mouse position into a numeric value. Sliders can be one- or two-dimensional, returning one or two coordinates. One-dimensional sliders can be horizontal or vertical. A scroll bar is a form of slider. Adjusting the granularity can turn a slider into a multi-position switch.

All the slider resources are optional. However at least one of the *ValueX* or *ValueY* resources must be present.

ValueX

The slider's X value. This is a value between 0 and 1. When the cursor is at the left edge of the viewport, *ValueX* is 0. When it is at the right edge, it equals 1.

ValueY

The slider's Y value. This is a value between 0 and 1. When the cursor is at the bottom edge of the viewport, *ValueY* is 0. When it is at the top edge, it equals 1.

ActiveArea

The screen cursor must be within this polygon for the slider to react to user input.

Start

This resource identifies a marker object that is placed at the lower limit of the X and Y values. For example, in a horizontal slider, the start marker is at the left edge of the range, while in a two-dimensional slider, it is placed at the lower left corner.

XEnd

This resource identifies a marker object that is placed at the upper limit of the slider X value.

YEnd

This resource identifies a marker object that is placed at the upper limit of the slider Y value.

Granularity

A control's **granularity** is the number of possible positions that control can take. This resource is an integer indicating that value. As an example, a granularity of 2 for a vertical slider creates a 2-position linear switch. If the resource is absent, the slider may take any value between the lower and upper limits.

DisableMotion

If this resource is present and non-zero, the control is disabled.

IncrementOnClick

If this resource is not present or is set to zero, the slider moves to the location of the mouse click. If the resource is present and non-zero, the alternate behavior is used for mouse clicks outside of the slider's *ActiveElement*. Each click moves the slider by its *PageIncrement* in the direction of the mouse. If the mouse button is held down, the slider keeps moving until the button is released or the slider reaches the mouse position.

RepeatTimeout

Defines a timeout in seconds after which the slider with *IncrementOnClick* starts moving when the mouse button is held down. If the resource is not defined, the default value of 250 ms is used.

RepeatInterval

Defines how fast a slider with *IncrementOnClick* repeats the slider movement when the mouse button is held down. If the resource is not defined, the default value of 100 ms is used.

Stateless

If this resource is present and non-zero, the slider has no state. That is, each time a move operation is completed, the slider values moves back to the center of their range, and delivers the final move coordinates to the application program via the message object. The view sliders in the GLG Graphics Builder are stateless. This allows the sliders to control an unlimited range.

Plane

The slider element appears to slide on the plane defined by this polygon. The points of the polygon must be co-planar. When you click on the slider widget, the cursor position is projected onto this plane. The resulting coordinates are used to set the slider position.

Increment

Increment for changing the knob's value by using the directional buttons listed below. The increment is expressed as a fraction of the total range of the slider (ranging from 0 to 1). If this resource is missing, the default increment is two hundredth of the slider range.

PageIncrement

Page increment for changing the knob's value by using the page directional buttons listed below. The increment is expressed as a fraction of the total range of the slider (ranging from 0 to 1). If this resource is missing, the default increment is one tenth of the slider range.

Left, Right, Up, Down

If buttons with these names are embedded into a two-dimensional slider, each press of a button moves the slider in the direction indicated by the button's name and by the amount specified by the slider's *Increment*.

Increase, Decrease

If buttons with these names are embedded into a one-dimensional slider, each press of a button moves the slider in the direction indicated by the button's name and by the amount specified by the slider's *Increment*.

IncreaseKeys, DecreaseKeys

These S resources define a list of characters that will be used as keyboard accelerators for incrementing or decrementing the slider's value.

PageIncrease, PageDecrease

If buttons with these names are embedded into a one-dimensional slider, each press of a button moves the slider in the direction corresponding to the button's name and by the amount specified by the slider's *PageIncrement*.

PageIncreaseKeys, PageDecreaseKeys

These S resources define a list of characters that will be used as keyboard accelerators for incrementing or decrementing the slider's value by *PageIncrement*.

Wrap

If this resource is present and non-zero, the slider will wrap around when the value is incremented or decremented past its low or high values.

SliderSize, StartPosition, EndPosition

These resources are defined in most of the slider and scrollbar objects to control the size of the slider's *ActiveElement* and the extent of its movement. These resources are used only to define the slider's geometry and are not used by the *GlgSlider* interaction handler. These resources are not present in the slider's message object and may be accessed only as resources of the slider's viewport.

Messages

The *GlgSlider* interaction handler supports the following messages that can be sent using the `GlgSendMessage` method:

Increase

Increases the slider's value by its *Increment*. The message has no parameters.

Decrease

Decreases the slider's value by its *Increment*. The message has no parameters.

PageIncrease

Increases the slider's value by its *PageIncrement*. The message has no parameters.

PageDecrease

Decreases the slider's value by its *PageIncrement*. The message has no parameters.

Up

Increases the slider's Y value by its *Increment*. The message has no parameters.

Down

Decreases the slider's Y value by its *Increment*. The message has no parameters.

Right

Increases the slider's X value by its *Increment*. The message has no parameters.

Left

Decreases the slider's X value by its *Increment*. The message has no parameters.

UpLeft, UpRight, DownLeft, DownRight

Composite messages that perform the actions of the two messages indicated by their names. The messages have no parameters.

GlgNSlider

The GLG Toolkit includes a native slider input handler that uses features of the native windowing environment and may be attached to a viewport object with *WidgetType* of VERTICAL_SCROLLBAR, HORIZONTAL_SCROLLBAR, VERTICAL_SCALE and HORIZONTAL_SCALE. The native slider handler encapsulates the native slider and scrollbar widget's interfaces and allows to handle them in a cross-platform way. The native slider is more limited than the GLG version, and only handles one-dimensional input. Its *Value* and *Stateless* resources are described in the *GlgSlider* section, but it also has the following resources:

Increment

Specifies the amount the value changes when the users moves the slider by one increment. The increment is expressed as a fraction of the total range of the slider (ranging from 0 to 1). If this resource is missing, the default increment of the native slider widget is used.

PageIncrement

Specifies the amount the value changes when the user moves the slider by one page increment. The page increment is expressed as a fraction of the total range of the slider (ranging from 0 to 1). It can also be set to -1, in which case the slider size will be used as a page increment. If the page increment is missing, a default page increment is used.

In the C# environment, .NET scrollbar controls do not allow setting the page increment and slider size independently. If *PageIncrement* is set to a positive value, it will define both the page increment and the slider size, otherwise *SliderSize* will be used to define both parameters.

SliderSize

Specifies the size of the moving part of a scrollbar. The slider size is expressed as a fraction of the total range of the slider (ranging from 0 to 1). If slider size is missing, the default size is used. This resource is applied only to the native scrollbar widget.

Granularity

Specifies a number of possible positions the slider can take. If the resource is absent, the slider may take any value between the lower and upper limits.

DrawTicks (Java only)

Activates display of the JSlider's major ticks if set to an integer value greater than 0. If the *Granularity* resource is not defined, the number of ticks is defined by the value of the *DrawTicks* resource. If *Granularity* is defined, its value is used as the number of ticks.

Messages

The *GlgNSlider* interaction handler supports the following messages that can be sent using the `GlgSendMessage` method:

Increase

Increases the slider's value by its *Increment*. The message has no parameters.

Decrease

Decreases the slider's value by its *Increment*. The message has no parameters.

PageIncrease

Increases the slider's value by its *PageIncrement*. The message has no parameters.

PageDecrease

Decreases the slider's value by its *PageIncrement*. The message has no parameters.

GlgKnob

A knob input widget is used to translate the angular position of the mouse into a value between 0 and 1. Adjusting the granularity can turn a knob into a multi-position switch. All knob angles, like all angles in a GLG drawing, are measured in degrees from the X axis (the 3 o'clock position).

All the knob resources are optional, except for *Value*.

Value

The knob's value. This is a number between 0 and 1. The knob is at 0 when the cursor is at the *StartAngle* position relative to the *Center*, and 1 when the cursor is at the *EndAngle* position. None of these resources need be present, in which case, the start angle is 0, the end angle is 360, and the center is the origin of the viewport.

Center

The angular position of the cursor is measured relative to this point. This resource is actually a marker object, which may or may not be visible, but in either case is used to define a position in the viewport space. If the resource is not present, the viewport's origin is used as the center.

StartAngle

This is the angle, measured counter-clock wise in degrees from the 3 o'clock position, at which the knob value is 0. If absent, the default value of 0 is used.

EndAngle

This is the angle, measured counter-clock wise in degrees from the 3 o'clock position, at which the knob value is 1. If absent, the RotateAngle value is used. The default value of 360 is used if the *RotateAngle* is absent.

RotateAngle

An alternative rotation angle measured counter-clock wise in degrees from the 3 o'clock position and relative to the *StartAngle*. If the *EndAngle* is absent, and the *RotateAngle* is defined, its value is used to define the end angle of rotation as *StartAngle* + *RotateAngle*.

Granularity

A control's granularity is the number of possible positions that control can take. This resource is an integer indicating that value. As an example, a granularity of 3 for a knob creates a 3-position rotary switch. If the resource is absent, the knob may take any value between the lower and upper limits.

DisableMotion

If this resource is present and non-zero, the control is disabled.

IncrementOnClick

If this resource is not present or is set to zero, the knob moves to the location of the mouse click. If the resource is present and non-zero, the alternate behavior is used for mouse clicks outside of the knob's *ActiveElement*. Each click moves the knob by its *PageIncrement* in the direction of the mouse. If the mouse button is held down, the knob keeps moving until the button is released or the knob reaches the mouse position.

RepeatTimeout

Defines a timeout in seconds after which the knob with *IncrementOnClick* starts moving when the mouse button is held down. If the resource is not defined, the default value of 250 ms is used.

RepeatInterval

Defines how fast a knob with *IncrementOnClick* repeats the movement when the mouse button is held down. If the resource is not defined, the default value of 100 ms is used.

Stateless

If this resource is present and non-zero, the knob has no state. That is, each time a move operation is completed, the knob *Value* moves back to the center of the knob range, and delivers the final move coordinates to the application program via the message object.

Plane

The knob element appears to rotate on the plane defined by this polygon. The points of the polygon must be co-planar. When you click on the knob widget, the cursor position is projected onto this plane. The resulting coordinates are used to set the knob position.

Increment

Increment for changing the knob's value by using Increase and Decrease buttons. The

increment is expressed as a fraction of the total range of the knob (ranging from 0 to 1). If this resource is missing, the default increment is one tenth of the knob range.

Increase, Decrease

If buttons with these names are embedded into a knob viewport, each press of a button changes the knob's value in the direction indicated by the button's name and by the amount defined by its *Increment*.

IncreaseKeys, DecreaseKeys

These S resources define a list of characters that will be used as keyboard accelerators for incrementing or decrementing the knob's value.

PageIncrease, PageDecrease

If buttons with these names are embedded into a knob, each press of a button moves the knob in the direction corresponding to the button's name and by the amount specified by the knob's *PageIncrement*.

PageIncreaseKeys, PageDecreaseKeys

These S resources define a list of characters that will be used as keyboard accelerators for incrementing or decrementing the knob's value by *PageIncrement*.

Wrap

If this resource is present and non-zero, the knob will wrap around when the value is incremented or decremented past its low or high values.

Messages

The *GlgKnob* interaction handler supports the following messages that can be sent using the `GlgSendMessage` method:

Increase

Increases the knob's value by its *Increment*. The message has no parameters.

Decrease

Decreases the knob's value by its *Increment*. The message has no parameters.

PageIncrease

Increases the knob's value by its *PageIncrement*. The message has no parameters.

PageDecrease

Decreases the knob's value by its *PageIncrement*. The message has no parameters.

GlgButton

A button widget reacts to left mouse button clicks while the mouse cursor is within the widget viewport. There are two kinds of buttons: toggle buttons and push buttons. A toggle button has an internal state that changes with each press of the button, while a push button has no internal state,

and only reacts to the external event (the mouse click). A toggle button's state may or may not be displayed. *GlgButton* supports only binary toggle buttons. A slider or knob with a non-zero *Granularity* resource can be used to create multi-position switches.

PressedState

This resource is usually 0, but when the mouse button is pressed, the resource momentarily changes to 1. When the button is released, the resource goes back to 0.

OnState

If this resource is present, the button is a toggle button, and successive clicks on the button change this resource value from 0 to 1 and back again. The value is set to 0 at startup and after a drawing reset.

InState

This resource is usually 0, but when the mouse cursor enters the button widget viewport, it changes to 1. When the cursor exits the viewport, the value goes back to 0.

Label

This resource indicates a text object displaying a label for the button.

LabelString

This resource generally indicates the string displayed by the *Label* resource text object. It is used when the button is displayed as part of a menu widget.

TooltipString

When the cursor enters the button viewport, and is still briefly, a small label appears displaying the string indicated by this resource. The *ButtonTooltip* event is generated when a button tooltip is activated or erased. Refer to the *Tooltip Message Object* section on page 364 of the *GLG Programming Reference Manual* for more details.

RepeatTimeout

Defines an interval in seconds after which the button starts generating repeated *Activate* messages if the button is held down. If it is set to a value less or equal to 0 (default value), the button repeat is disabled. Repeated action is enabled only for push buttons with *ActOnPress* activation.

RepeatInterval

Defines an interval in seconds between repeated *Activate* messages generated when the button is held down. The default value is 1/10 of a second.

ActOnPress

If this resource is present and has a non-zero value, the button's action happens on the down-click of the mouse button. Otherwise, the action is taken on the release of the button. If this resource is not present, and if the mouse cursor is moved out of the button viewport before the mouse button is released, that no action is taken.

ArmedState

If this resource is present and has a value different from -1, the button is enabled (armed) only when the *Control* key is pressed. Without the *Control* key, the button is locked. This type of buttons is used in a process control applications where it is important to prevent the user from triggering an action by accidentally pressing a button. When the *Control* key is pressed, the

resource changes to 1, returning to 0 when the *Control* key is released.

If the resource is set to -1, the *ArmedState* functionality is disabled and the button does not pay attention to the *Control* key. This makes it possible to use a single button widget template and enable or disable the *ArmedState* functionality for individual button instances.

TokenValue

The button's "value" is an arbitrary scalar value assigned to the button. This resource is used when the button is part of a menu widget.

Messages

The *GlgButton* interaction handler supports the following messages that can be sent using the `GlgSendMessage` method:

Set

Sets the toggle (toggle buttons only). The message has no parameters.

Reset

Resets the toggle (toggle buttons only). The message has no parameters.

Activate

Generates an Activate event (push buttons only). The message has no parameters.

GlgNButton

The GLG Toolkit provides a native button input handler which may be attached to a viewport object with *WidgetType* of PUSH_BUTTON, CUSTOM_BUTTON and TOGGLE_BUTTON. The native button handler encapsulates the native push button, toggle and checkbox widget's interface and allows to handle native buttons in a cross-platform way.

Its only resources are *PressedState*, *OnState*, *LabelString*, *TooltipString*, *RepeatInterval* and *RepeatTimeout*, with the same meaning as for the *GlgButton* handler. Note that, whereas the GLG button widget's label is supplied by a text object, the native widget uses a simple string value, and draws the label itself.

The *OnState* resource is supported not only for the TOGGLE_BUTTON, but also for other button types. If the *LabelString* resource of a push button has the list transformation attached, the *OnState* resource may be used to control the list transformation to toggle the button's label on a mouse click. Such button will behave as a toggle button, but without rendering the toggle button's state indicator. The button will still generate the push button's *Activate* action.

Messages

The *GlgNButton* interaction handler supports the same messages as the *GlgButton* handler listed above.

GlgNText

This is a native text widget handler which may be used with both single and multi-line text widgets. It may be attached to a viewport object with *WidgetType* of TEXT and TEXT_EDIT. The native text handler encapsulates the native text box widget's interface and allows to handle it in a cross-platform way.

TextString

This resource indicates the string attribute of the text object.

MaxLength

This optional resource indicates the maximum length of a text string that user can enter from the keyboard. This resource does not affect strings that are entered programmatically by setting the *TextString* resource.

InputFormat

This S-type resource indicates an optional format and may have the following values:

- "string" - for entering any alpha-numerical characters
- "integer" - for entering integer values
- "double" - for entering floating-point values

If the resource is not specified, the "string" default value is assumed.

MinValue, MaxValue

Specify optional minimum and maximum values of the numerical input.

EnforceRange

For the numerical input, this optional resource defines how the values outside of the [*MinValue*, *MaxValue*] range are handled. If it is defined and set to 0, the input values will not be adjusted and may exceed the range. If it is set to 1 or not defined, the input values outside of the range will be adjusted to fit inside the range.

InputInvalid

This resource contains the input status for numerical inputs, and may be set to the following values when the input is completed:

- 0 - input was parsed successfully
- 1 - input parsing error
- 2 - input our of range

Value

This resource contains the entered value of the numerical input object.

ValueFormat

This resource specifies an optional C-style format that controls how the entered value is displayed in the numerical input object.

GlgText

The text widget is used for entering single lines of typed text. It also contains a resource that provides initial text to display, and another to control the widget's appearance when it receives the input focus. This handler is superseded with the *GlgNText* and is maintained for backward compatibility.

TextObject

This resource indicates the scrolled text object where the text is to be typed.

TextString

This optional resource indicates the string attribute of the text object.

Focus

This resource changes when the widget is available for input. It is normally 0, but changes to 1 when you click on the text widget with the mouse. It is used to control the look of the widget when it is ready to accept typed input. For example, you could use a linear transformation to make a border around the widget appear when the widget is selected.

GlgSpinner

A spinner displays a numerical value and contains two or more buttons to increase or decrease it. A text edit box widget may be used to display and edit the value, or the value may be presented as a display-only text object which could only be altered by a predefined increment using the increase and decrease buttons. Some of the spinner's resources may be inherited from the *GlgText* handler of the embedded text edit box used to display the spinner's value. In this case, aliases are used to "redirect" the resource by pointing to the corresponding resource of the embedded text widget. An optional slider widget may be used to implement a "sliding" spinner which allows the user to change its value using either a text edit box, a slider, or increase and decrease buttons.

Value

The spinner's value.

MinValue, MaxValue

The spinner's minimum and maximum values (optional).

Wrap

If this resource is present and non-zero, the spinner will wrap around when the value is incremented or decremented past its low or high values.

Increment

Increment for changing the value by using *Increase* and *Decrease* buttons.

PageIncrement

Increment for changing the value by using *PageIncrease* and *PageDecrease* buttons.

Increase, Decrease

If buttons with these names are embedded into a spinner, each press of a button changes the spinner's value in the direction indicated by the button's name and by the amount defined by

its *Increment*.

PageIncrease, PageDecrease

If buttons with these names are embedded into a spinner viewport, each press of a button changes the spinner's value in the direction corresponding to the button's name and by the amount defined by its *PageIcrement*.

IncreaseKeys, DecreaseKeys

These S resources define a list of characters that will be used as keyboard accelerators for incrementing or decrementing the slider's value by its *Increment*.

PageIncreaseKeys, PageDecreaseKeys

These S resources define a list of characters that will be used as keyboard accelerators for incrementing or decrementing the slider's value by its *PageIncrement*.

TextInput

An optional text entry widget that may be used to display spinner's value.

Slider

An optional slider widget that may be used in a sliding spinner.

Done

An optional Done button for generating *Activate* message.

Messages

The *GlgSpinner* interaction handler supports the following messages that can be sent using the `GlgSendMessage` method:

Increase

Increases the spinner's value by its *Increment*. The message has no parameters.

Decrease

Decreases the spinner's value by its *Increment*. The message has no parameters.

PageIncrease

Increases the spinner's value by its Page*Increment*. The message has no parameters.

PageDecrease

Decreases the spinner's value by its *PageIncrement*. The message has no parameters.

GlgNList

The native list handler encapsulates the behavior of a native list widget, allowing to use it in a cross-platform way. It may be attached to a viewport object with the *WidgetType* of LIST, MULT_LIST and EXT_LIST, and handles both the single and multiple selection, depending on the native list type. In the single selection mode, the list's selection may be changed by setting the value of the *SelectedIndex* resource. In the multiple selection mode, the *GlgSendMessage* method may be used to change list's selection as well as to add, delete or query list entries.

InitItemList

This resource is a list of strings to be displayed in the list widget on initial appearance. It may be edited in the Graphics Builder.

ItemList

A group object created by the list handler which contains the current list of strings displayed in the list widget.

SelectedIndex

In the single selection mode, the value of this resource is set to the 0-based index of the selected list item.

SelectedItem

In the single selection mode, the value of this resource is set to the string of the selected list item.

ItemStateList

In the multiple selection mode, the list creates this resource to hold the selection state of its items. The resource is a group object containing integer values.

Messages

The *GlgNList* interaction handler supports the following messages that can be sent using the `GlgSendMessage` method:

SetInitItemList

Updates the list widget with the new items from the viewport's *InitItemList* after changing items of the *InitItemList* resource. The message has no parameters.

SetItemList

Updates the list widget with the items from the item list passed as the first parameter of the message. The passed item list must be a group object containing item strings. The message does not alter *InitItemList*.

GetItemList

Returns a current list of items displayed in the list widget. This message has no parameters and returns a group object containing a list of strings.

AddItem

Adds a new item to the list. The new list item is passed as the first message parameter, and the second parameter may contain GLG_TOP or GLG_BOTTOM to specify the place to add the item to. If the second parameter is NULL, the default GLG_BOTTOM value is used. The *UpdateItemList* message must be used to display the new items when finished. The *SetItemList* message provides a way to replace the whole item list.

DeleteItem

Deletes a list item. The first message parameter may contain GLG_TOP or GLG_BOTTOM to specify the item to be deleted. If the parameter is NULL, the default GLG_BOTTOM value is used. The *UpdateItemList* message must be used to update display when finished. The *SetItemList* message provides a way to replace the whole item list.

UpdateItemList

Updates the list's display after changing its *ItemList*. The message has no parameters.

GetItemCount

Returns a number of items in the list. This message has no parameters.

SetItemState

Sets the item specified by the zero-based index passed as the first parameter of the message to the state (*True* or *False*) specified by the second message parameter.

GetItemState

Returns the state of the item specified by the zero-based index passed as the first parameter of the message.

SetItemStateList

Sets the state of all items of a multiple-selection list. The list of new item states is supplied as the first parameter of the message and must be a group object containing integer values. The number of items in the list must match the number of displayed items in the list widget.

GetItemStateList

Returns a list of item states of a multiple-selection list widget. The list of states is returned as a group object containing integer values. This message has no parameters.

ResetAllItemsState

Deselects all selected items. This message has no parameters.

GetSelectedItemList

Returns a list of selected items. This message has no parameters and returns a group object containing integer values.

GlgNOption

The native option menu handler encapsulates the behavior of native option menu (X Windows) and combo box (Windows) widgets, allowing to use them in a cross-platform way. It may be attached to a viewport object with the OPTION_MENU *WidgetType*. The option menu's selection may be changed by setting the value of the *SelectedIndex* resource. The *GlgSendMessage* method may be used to add, delete or query option menu's entries.

InitItemList

This resource is a list of strings to be displayed in the option menu or combo box widget on initial appearance. It may be edited in the Graphics Builder.

ItemList

A group object created by the option menu handler which contains the current list of option strings displayed in the widget.

InitSelectedIndex

This optional resource (D type) provides a zero-based index of an item to be selected on the initial appearance of the option menu.

SelectedIndex

The value of this resource is set to the 0-based index of the selected option item when the selection is made. Setting this resource from a program changes the displayed selection.

SelectedItem

The value of this resource is set to the string of the selected option item.

Messages

The *GlgNOption* interaction handler supports the following messages that can be sent using the `GlgSendMessage` method:

SetInitItemList

Updates the option menu widget with the new option items from the viewport's *InitItemList* after changing items of the *InitItemList* resource. The message has no parameters.

SetItemList

Updates the option menu widget with the option items from the item list passed as the first parameter of the message. The passed item list must be a group object containing item strings. The message does not alter *InitItemList*.

GetItemList

Returns a current list of option menu items. This message has no parameters and returns a group object containing a list of strings.

AddItem

Adds a new option menu item to the widget. The new item is passed as the first message parameter. The second parameter may contain GLG_TOP or GLG_BOTTOM to specify the place in the list of options where the new item will be added. If the second parameter is NULL, the default GLG_BOTTOM value is used. The *UpdateItemList* message must be used to display the new option items when finished. The *SetItemList* message provides a way to replace the whole option list.

DeleteItem

Deletes an option item. The first message parameter may contain GLG_TOP or GLG_BOTTOM to specify the option item to be deleted. If the parameter is NULL, the default GLG_BOTTOM value is used. The *UpdateItemList* message must be used to update display when finished. The *SetItemList* message provides a way to replace the whole item list.

UpdateItemList

Updates the widget's display after changing its *ItemList*. The message has no parameters.

GetItemCount

Returns a number of option menu items. This message has no parameters.

GlgMenu

A menu widget is a container used to manage a set of buttons. When a button within a menu is pressed, it issues an *Activate* message object. The menu converts this *Activate* message of the button into an *Activate* message for the menu, supplying the logical button number in the message. This

allows an application program to ignore the presence or absence of specific buttons in the menu and to deal only with a menu as a whole. The menu widget also supplies information about the selected button's label and assigned value by placing this information into the menu's message object. The buttons placed in the menu are recognized by their names. The name of the first button must be *Button0*, the name of the second *Button1*, and so on.

Buttons may be pasted into the menu and positioned inside it using the GLG Graphics Builder. Alternatively, a series object may be used to create a required number of buttons from the series' template, as shown in the menu widgets provided in the Special Widgets set. With either method, a menu widget manages the geometry of the buttons, resizing them proportionally when the menu is resized.

If a menu contains toggle buttons (buttons with *OnState* resource) and one of the *SelectedIndex*, *SelectedString* or *SelectedValue* menu resources are defined, the menu will behave as a radio menu, allowing the user to select only one toggle button. In this case, the *SelectedIndex* resource may be set to a value of a zero-based button index to select the corresponding button. If none of the selection state resources are defined, the menu will behave as a multiple-selection menu.

Menu button labels may be assigned dynamically by changing the *String* attribute of a button's label. Button labels may also be applied with the *LabelList* resource. If present, this resource identifies a list of data properties of S type specifying labels to use (usually attached to the menu's viewport as a Custom Property list). The buttons in the menu will be assigned the labels taken from these objects. If there are more buttons then the labels in the *LabelList*, the *LabelString* of the button itself is used. If there are more labels in the group than there are buttons in the menu, the menu may be scrolled (an explicitly defined scrollbar named *ScrollObject* must be provided to facilitate scrolling).

Button<n>

The menu buttons. The first button is called *Button0*, and must be present for any menu widget. The other buttons in the menu must follow this first button in sequence. The numeric suffix is the index of the button, and is returned with the *SelectedIndex* resource of the message object.

LabelList

An optional group object containing a list of the button labels (S data objects) to be displayed. If no such resource exists, the button's name or LabelString resource is used. To redraw button labels after changing strings in the list, reset the menu.

TooltipList

An optional group object containing a list of the tooltip strings (S data objects) used to initialize the buttons on hierarchy setup.

InitStateList

An optional group object containing a list of initial state values (D data objects) used to set the initial state of toggle buttons of a multiple-selection menu.

InitSelectedIndex

This optional resource (D type) provides a zero-based index of a toggle button to be selected on the initial appearance of a radio menu.

ScrollObject

If the list is too long for the menu window, a scroll object may be included. This is a vertical slider widget.

SelectedIndex, SelectedString, SelectedValue

These resources may be present in the menu viewport. They contain the index, label string, and token value of the last button pressed. They are more often used as resources of the message object returned when a menu button is pressed. However, the *SelectedIndex* resource may be used for selecting an item of a radio menu at run time by setting the resource to a zero-based button index.

Messages

The *GlgMenu* interaction handler supports the following message that can be sent using the `GlgSendMessage` method:

GetItemCount

Returns the number of buttons in the menu. The message has no parameters.

GlgBrowser and GlgFontBrowser

The browsers are an arrangement of menus, buttons and lists designed to facilitate the selection of some item. Typical arrangements include a **filter**, or string containing wildcard characters, a list of objects that match that string, and a text widget into which a user can type the selection. Often, the filter string will be reproduced in the selection text widget to save typing by the user.

Different browsers include optimizations designed for the objects being browsed. A font browser, designed for making selections from fonts that use the X font naming conventions, has a menu of buttons designed to make the construction of a filter string easy. The font browser is only supported on X Windows.

The browser handler is designed to be used for browsing resources of GLG drawings and objects, a example of which can be seen as the resource browser in the GLG Graphics Builder. The browser handler may also be used to browse tags (tag browser) and custom data sources (custom data browser).

Menu

The menu from which selections may be made. For a resource browser, the menu presents the resources on the chosen level of the hierarchy. For a tag browser, the menu displays a list of tags of the selected object. For a custom data browser, the menu displays a list of data sources. For a font browser, it displays either fonts matching the filter string or a list of available entries for a specified filter field. The menu object uses a native list widget (a viewport with *WidgetType*=LIST).

Filter

This text widget displays the filter string used to display the entries in the menu. Only entries that match the filter are displayed.

Selection

This text widget displays the selection that will be made. To save typing, the browser usually echoes the filter string into this widget.

FontName (Font Browser only)

This is a string resource containing the name of the font chosen.

FontSampleName (Font Browser only)

This text object is used to display a sample of the chosen font. Its *String* resource is usually some sample sentence.

FieldMenu (Font Browser only)

This resource indicates a menu used to specify the field of the font filter string a user wishes to edit. That is, if you want to change the font family, you push the *Family* button in this menu. The browser then presents you with the available options. When you choose one of them, the filter is modified accordingly.

TypeObject

This resource is a string value defining the type of the browser for the *GlgBrowser* input handler, not used by the *GlgFontBrowser*. The value of the string must be "*Resource*" for the resource browser, "*Tag*" for the tag browser and "*Data*" for the custom data browser.

The browser has three buttons, to signal the action to take. For details of the messages these buttons send, see the description of the *GlgButton* and *GlgNButton* handlers. The button messages are processed by the browser handler and do not require any additional handling.

Done

Pushing this button indicates that the selection has been made.

Cancel

Pushing this button sends a *Cancel* message, which may be used to erase the browser, without making a selection.

Reset

This button resets the filter to its default value.

GlgPalette

The palette widget is used to present a user with the opportunity to select one from a variety of objects. This widget is designed to be used within a program using the GLG API.

A palette widget has only one resource, an object named *PaletteObject*. This is a container object with several members, each corresponding to a different possible choice. The user selects one of the presented objects with the left mouse button, and the chosen object is returned to the calling program in the callback message object. The returned object is indicated with the *SelectedObject* resource of the message object.

PaletteObject

A container object presenting a variety of objects to be chosen by a user.

GlgClock

The clock input handler is used for two different timekeeping tasks: measuring elapsed time, and displaying the absolute time. Unlike the other input handlers, the clock widget is not primarily used for user input. Rather, it is used for updating a drawing with the current time.

If the *TimerState* resource is present, the widget becomes a stopwatch, measuring elapsed time instead of absolute time. Used as a stopwatch, the handler recognizes user input when the control buttons are operated. All the resources are optional.

Hour, Min, Sec

These resources indicate the time in its traditional format. All scalar data types, the hour ranges from 0 to 11, and the minute and second resources range from 0 to 59.

ValueHour, ValueMin, ValueSec

These resources are also used to indicate the time. However, instead of measuring hours, minutes, and seconds, they indicate the proportion of the clock's circumference used to indicate the desired quantity. For example, six hours is indicated by the value 0.5, and 15 seconds by the value 0.25. These resources are generally used to drive the arms of a clock drawing.

TimeString

This string resource contains a character string indicating the time.

TimerState

If this resource is present, the widget functions as a stopwatch. If the clock is running, the *TimerState* is 1, otherwise it equals 0.

Start, Stop, Reset

These three resources each indicate a button widget. When the *Start* button is pushed, the stopwatch begins timing. The *Stop* button stops the clock, and the *Reset* button resets the time to zero.

Messages

The *GlgClock* interaction handler with the *TimerState* resource 9stopwatch) supports the following messages that can be sent using the `GlgSendMessage` method:

Start

Starts the stopwatch. The message has no parameters.

Stop

Stops the stopwatch. The message has no parameters.

Reset

Resets the stopwatch. The message has no parameters.

Native Widgets

In addition to making drawings using GLG widgets, you can use the GLG Toolkit to create an arrangement of widgets native to the windowing environment. The type of native widget used to render a viewport is defined by the *WidgetType* viewport attribute. This attribute is inherited from the screen object associated with each viewport, and available for editing in the Builder by clicking on the *More* button in the list of viewport properties.

The default DRAWING_AREA widget type is used to render viewports that are used as drawing surfaces for displaying graphical objects. By placing viewport objects with widget type different from the default into the GLG drawing, you can embed native widgets into the drawing. The look and feel of the native widgets will change to match the environment in which the drawing is displayed. For example, a native button will be displayed as a Windows, Motif, or Swing button when it is displayed in the respective environments.

Native widgets come with some limitations. For example, you cannot use the drawing resources to control the behavior of these widgets, although you can control graphical features such as color, layout and, via input handler's resources, labels. The input handlers are provided only for some of the native widgets: buttons, toggles, sliders, scrollbars, text field, list and option menu widgets.

For complete control over a native widget's resources, in both C/C++ , Java and C#/.NET, use the windowing environment's native API. The Builder's graphical interface is still quite useful for application design tasks such as arranging windows and graphical controls.

The **Widget ID** of the native widget used for rendering the viewport may be obtained by querying the *Widget* resource of the viewport using the *GlgGetLResource* method of the GLG API (similar to the *GlgGetDResource* method, but uses *long* return value type; *GetResource* method returning *Object* is the Java and C#/.NET equivalent). The Widget ID must be queried after setting up the drawing hierarchy (at which time the native widget is created) and may be used in any further native API calls.

The widget types are listed in the table below, with their X Windows/Motif, Windows, Java and .NET equivalents. To learn how to control these widgets, refer to the documentation for the appropriate windowing environment.

Widget Types and Their Equivalences

GLG *WidgetType*	**X Windows/Motif Widget Type**	**Windows Window Class**	**Java Component Class**	**.NET Component Class**
Drawing Area	*XmDrawingArea*	WINDOW	AWT: Panel Swing: JPanel	UserControl
Push Button	*XmPushButton*	Push style BUTTON	AWT: Button Swing: JButton	Button
Toggle Button	*XmToggleButton*	Toggle Style BUTTON	AWT: Checkbox Swing: JCheckBox	CheckBox
Custom Button	*XmDrawnButton*	WINDOW	AWT: Panel Swing: JPanel	UserControl

Widget Types and Their Equivalences

GLG *WidgetType*	**X Windows/Motif Widget Type**	**Windows Window Class**	**Java Component Class**	**.NET Component Class**
Arrows (Left, Right, Up, and Down)	*XmArrowButton*	Push style BUTTON with Left, Right, Up or Down label.	AWT: Button with a label Swing: JButton with a label	Button with a label
Scale (Horizontal, Vertical)	*XmScale*	SCROLLBAR	AWT: Scrollbar Swing: JSlider	HScrollbar or VScrollbar
Scrollbar (Horizontal, Vertical)	*XmScrollBar*	SCROLLBAR	AWT: Scrollbar Swing: JScrollBar	HScrollbar or VScrollbar
Text	*XmTextField*	EDIT (single line)	AWT: TextField Swing: JTextField	TextBox
Text Edit	*XmText*	EDIT (multi-line)	AWT: TextArea Swing: JTextArea	TextBox
Label	*XmLabel*	Text style STATIC	AWT: Label Swing: JLabel	Label
Option Menu	*XmOptionMenu*	Drop-down List style COMBOBOX	AWT: Choice Swing: JComboBox	ComboBox
Separator	*XmSeparator*	WINDOW	AWT: Panel Swing: JPanel	UserControl
List	*XmList*	LISTBOX	AWT: List Swing: JList	ListBox
Bulletin	*XmBulletinBoard*	WINDOW	AWT: Panel Swing: JPanel	UserControl
Form	*XmForm*	WINDOW	AWT: Panel Swing: JPanel	UserControl
Dialog Area	*XmDrawingArea*	WINDOW	AWT: Panel Swing: JPanel	UserControl

If you are using the Xt Wrapper Widget instead of the Motif-based Wrapper Widget, the Drawing Area widget is of the *Composite* class, and the other Motif native types are not available.

The user interface controls widgets, such as PUSH_BUTTON, SCROLLBAR, TEXT, LIST and others, are used in conjunction with the corresponding interaction handler (described earlier in this chapter) which handles user interaction with the native widget. On Windows, the color of these widgets is defined by the system's color settings, and the *FillColor* of the viewport is ignored. In Java, the color of these widgets is defined by the chosen Look and Fill scheme, and the viewport's *FillColor* is ignored as well. In UNIX / X Windows environment, the background color of the widgets is defined by the viewport's *FillColor*, and the widgets' forecolor is taken from the system's settings. The CUSTOM_BUTTOM widget is an exception, it uses viewport's *FillColor* as a background color in X Windows and Java environments, and system color on Windows.

The DIALOG_AREA widget may be used to provide a matching dialog background color for other native control widgets. On Windows, the color of the dialog widget is defined by the system color. In X Windows and Java environments, the color is defined either by the viewport's *FillColor*, or by the *GlgDefaultDialogColor* global configuration resource, which, if set, overrides the viewport's *FillColor*. This provides a global way to define the background of all dialogs in the X Windows and Java environments in a way similar to that on Windows. The default unset value of the *GlgDefaultDialogColor* global configuration resource is (-1, -1, -1).

When the DRAWING_AREA, CUSTOM_BUTTOM, BULLETIN, FORM and DIALOG_AREA widget types are used, the 3D bevels are drawn if the value of the *ShadowWidth* parameter is not equal 0.

Input Objects Design and the ValueParam resource

Starting with the release 3.4, the output resources of input handlers, such as *Value, ValueX, ValueY, OnState* and others, have a corresponding resource whose name is formed by adding the *"Param"* suffix: *ValueParam, ValueXParam, OnStateParam* and so on. These resources are used only in the internal design of the input widgets and can be ignored in the application code.

The following section provides detailed information on the internal design of input widgets; it is intended for system integrators who want to create custom input widgets from scratch. This information is not required for using the Control Widgets provided with the Toolkit and can be safely skipped.

Advanced: Internals of the Input Objects

Widgets that use *GlgSlider* and *GlgKnob* input handlers use *Move* and *Rotate* transformations to move the widgets' active elements. A *Range Conversion* transformation is attached to the *Factor* attribute of the *Move* and *Rotates* transformations; the *Input Value* parameter of the range conversion transformation controls the slider's or knob's active element and is named *Value, ValueX* or *ValueY* depending on the widget type.

The *Range Conversion* transformation converts a user-defined range of the slider or knob to the [0;1] range of the *Factor* attribute that drives the widget's active element. When a range transformation is attached, an interaction handler needs to know the object IDs of both the *Factor* and *Input Value* parameters in order to properly animate the widget. To accomplish that, the *Factor* attribute is named by adding the "*Param"* suffix to the name of the corresponding controlling variable: *ValueParam*, *ValueXParam* or *ValueYParam.*

To allow the input object's output resources to be constrained to other resources in the drawing without the loss of the input widget functionality, the *Value, ValueY* or *ValueY* resources are defined as aliases using default attribute names relative to the corresponding resource with the *Param* suffix. For example, in a horizontal slider, the *ValueX* resource is defined as an alias to *"ValueYParam/Xform/XformAttr6"*, which is relative to the *ValueYParam* resource. Since the default attribute names are not affected when the *ValueY* resource is constrained to a resource in the drawing with a different name, such constraining operation will not interfere with the functioning of the input handler. Previously, the resources in the drawing could be constrained to the input objects's output resources, but not visa versa.

To avoid resource name conflicts when *Value, ValueX* and *ValueY* resources are present in the drawing as both named resources and aliases, set *HasResources* of the active element or of the range transformation to make sure these resources are not visible at the top level directly, but only through the aliases.

For input widgets that do not have ranges, such as toggle buttons, the *Identity* transformation is used to enable the use of aliases in a similar way.

The new input objects were redesigned to support the functionality described above. The old widget design is still supported for backward compatibility.

Chapter 6

Using the GLG Graphics Builder

6

The GLG Graphics Builder is a tool that provides an interactive way to create or modify a GLG drawing. The Builder enables you to add new objects to a drawing, and to manipulate the graphical objects in the drawing. It provides access to the attributes of the graphical objects and their transformations. The Builder also includes a facility for testing the animation of a drawing.

Before you begin to use the Builder, we strongly recommend that you familiarize yourself with the structure and contents of GLG drawings; see the the *Structure of a GLG Drawing* chapter and the *GLG Objects* chapter. It is also recommended to go through the *GLG Builder and Animation Tutorial* to familiarize yourself with the basic editing operations.

This guide explains how to use the GLG Graphics Builder, outlining some of the basic tasks in creating and editing GLG drawings:

The GLG Builder also provides **scripting** capabilities for creating and editing drawings in a batch mode. Refer to the *GLG Programming Tools and Utilities* chapter of the *GLG Programming Reference* for details on using the Builder in scripting mode.

Creating a Drawing

Before you start the Builder, we suggest that you determine the general content and overall organization of the drawing you plan to create. Identify the parts you want to animate, and the resources you intend to name. Planning your drawing makes the drawing more efficient, and saves you time.

Viewing a GLG Drawing

A GLG drawing is an abstract hierarchy of objects. The hierarchy defines the relationships between the objects in a drawing and their attributes, which define their appearance and behavior.

When you open a GLG drawing in the Builder, the Builder reads the object hierarchy information in the drawing, using it to render a set of visible graphical objects in its drawing area. Because only part of the object hierarchy is composed of visible graphical shapes, the Builder can only present a partial view of the object hierarchy. The Builder shows the graphical shapes that make up the visible part of the drawing (polygons, lines, and the like) in its drawing area.

As you draw shapes in the Builder, set their attributes, and add transformations to them, you are constructing the hierarchy of objects that make up a drawing. The Builder uses dialogs to provide access to the other, non-graphical objects that are subordinate to the graphical objects, such as transformations and attributes.

In general, the Builder restricts the use of the term **object** to visible, selectable shapes that appear in the drawing area. An object immediately below such a shape in the object hierarchy is described as an **attribute** (though it is usually an object, too). An object may be defined as a **resource** by naming it. A member of an advanced object such as a group may be called a **subobject**. In the Builder, the words **attribute** and **property** are synonymous.

Viewing the Object Hierarchy

Because the Builder is a visual and interactive tool, it presents the drawing as a set of visible shapes that can be selected and edited. This focus on graphical objects — the shapes that make up the drawing — has the effect that the complete object hierarchy can be difficult to visualize in the Builder interface.

Although there is no single representation of the object hierarchy, several partial views are available:

- The dialogs for editing a graphical object's attributes and transformations provide access to an object's attributes as well as the attributes of the transformations attached to the object; see page 229.
- For a drawing with named objects, the Builder lets you browse a hierarchy of resources, very similar in appearance to a file and directory structure; see page 245.
- For a transformed object, the Builder lets you view the original, untransformed object. The drawing area shows a single level of the hierarchy of graphical objects, and you use *Traverse*, *Transformation Down* and *Traverse*, *Up* to move between the levels; see page 239.
- For a composite object such as a group or viewport, the Builder provides access to its subobjects; use *Traverse*, *Hierarchy Down* and *Traverse*, *Up* to move between the levels; see page 265.

Starting and Stopping the Builder

The *bin* subdirectory of the GLG installation contains executables of one or more GLG editors:

<glg>/bin/
    GlgBuilder
    GlgHMIEditor

If a GLG Editor is started from a command line, it starts in the OpenGL mode and uses the OpenGL driver, if it is available. The bin directory also contains links (shortcuts on Windows) for starting a GLG Editor in non-OpenGL (GDI) mode:

GlgBuilder_no_opengl
GlgHMIEditor_no_opengl

For the Enterprise Edition of the Graphics Builder, the bin directory also contains links for starting the Builder in the OEM mode:

GlgBuilder_OEM_no_opengl
GlgBuilder_OEM_no_opengl

Instead of using links, command options listed below may be used to specify the OpenGL or OEM mode.

To load a specific drawing file on startup, specify the drawing file name:

GlgBuilder *filename*

For **Windows** users, a GLG Editor may be started by choosing *GLG Graphics Builder* or *GLG HMI Configurator* from the *Start* Menu. The *Start* menu contains two separate folders: one for the OpenGL and another for the GDI version of the Toolkit.

Command-line Options

A number of command-line options and environment variables are also supported. The most common options are:

`-help`
: Prints all available command-line options on the terminal.
On Windows, writes command-line option information to the *glg_error.log* file. The location of the file is determined by the GLG_LOG_DIR and GLG_DIR environment variables as described in the *Error Processing* section on page 51 of the *GLG Programming Reference Manual*.

`-verbose`
: Generates diagnostic output for troubleshooting the OpenGL driver, editor setup, loadable editor extension DLLs and other editor extensions.

  On Windows, the output is saved in the *glg_error.log* file. The location of the file is determined by the GLG_LOG_DIR and GLG_DIR environment variables as described in the *Error Processing* section on page 51 of the *GLG Programming Reference Manual*.

`-config-file <filepath>`
: Specifies an alternative *glg_config* file to use.

`-oem`
: Starts the Enterprise Edition of the Graphics Builder in the OEM mode.

`-widget-editing-mode`
: In this mode, widgets loaded from the palettes with *Ctrl-click* can be saved into the original drawing files, facilitating convenient editing of widgets in the custom widget palettes. Without this option, a copy of a modified widget is saved in the current directory by default, to avoid permanently overwriting widgets in the GLG Builder palettes.

`-glg-disable-opengl`
: Disables OpenGL renderer in favor of the native windowing renderer.

`-glg-enable-opengl`
: Enables OpenGL renderer if present. The OpenGL renderer will be used only for the viewports which have their *OpenGLHint* attribute set to *ON*.

`-glg-opengl-version <NNN>`
: Specifies the value of *GlgOpenGLVersion* which requests the specified OpenGL version. The shader-based Core OpenGL profile is used for OpenGL versions higher than 3.00, and the Compatibility profile is used for older OpenGL versions. The Compatibility profile is used by default; an OpenGL version needs to be explicitly specified to use the Core profile.

`-glg-opengl-hardware-threshold <N>`
: Specifies the value of *GlgOpenGLHardwareThreshold*. All viewports with a non-zero value of the *OpenGLHint* attribute, less than or equal to *GlgOpenGLHardwareThreshold*, will be rendered using the hardware OpenGL renderer (if available). Viewports with the attribute value between *GlgOpenGLHardwareThreshold* and *GlgOpenGLThreshold* will be rendered using the software OpenGL renderer (if available).

`-glg-opengl-threshold <N>`
: Specifies the value of *GlgOpenGLThreshold*. All viewports with a value of the *OpenGLHint* attribute greater than *GlgOpenGLThreshold* will be rendered using the GDI renderer.

`-glg-disable-hardware-opengl`
: Disables hardware OpenGL. If the OpenGL driver is enabled, only the software-based OpenGL renderer will be used.

`-glg-disable-software-opengl`
: Disables software OpenGL. If the OpenGL driver is enabled, only the hardware-based OpenGL renderer will be used.

`-glg-debug-opengl`
: Generates extended diagnostic output for the OpenGL driver.

`-glg-disable-timers`
: Disables timer transformations in the drawings for debugging purposes.

Environment variables may be used instead of the command-line options if necessary.

A verbose mode may be set by setting the *GLG_VERBOSE* environment variable to *True*.

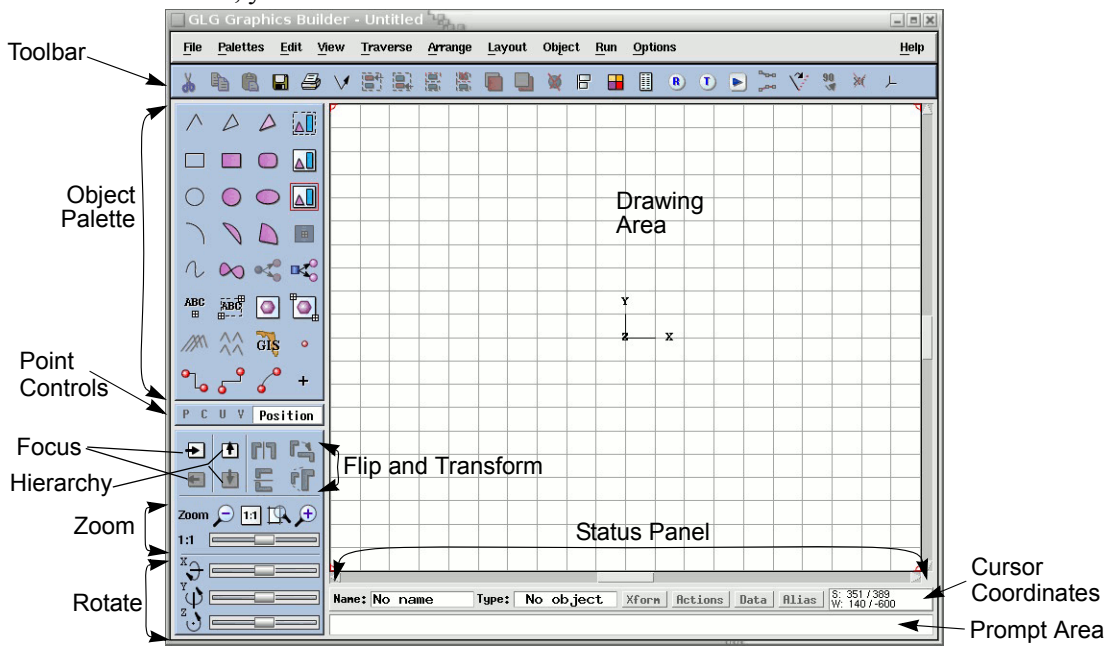
The OpenGL renderer may also be enabled or disabled by setting the *GLG_OPENGL_MODE* environment variable to *True* or *False*, or by setting the *GlgOpenGLMode* global configuration resource in the GLG configuration file to the following values:

0 - disable the OpenGL renderer
1 - enable the OpenGL renderer
-1 - don't change the default setting.

Refer to the *Builder Setup and Customization* chapter on page 279 for a list of all supported environment variables.

GLG Graphics Builder Features

When the Builder starts, you see the main window:



The most important areas of the GLG Graphics Builder window are:

The ***Toolbar*** contains icon buttons for fast access.

The ***Drawing Area*** contains the graphical objects of the drawing. By default, the drawing area contains an axis marker that shows the projection of the current view, and grid lines. The drawing area also contains round red markers that annotate the extent of the default span.

The markers are usually displayed in the corners of the viewport with the current editing focus, unless the viewport is zoomed in or out. The markers may be outside of the visible area if the viewport is zoomed in, of if *Stretch*=NO and *PushIn*=NO.

The ***Object Palette*** contains buttons for creating graphical objects.

The ***Control Panel*** contains the following controls:

- The ***Point Controls*** let you select a method for specifying the location of points as you draw an object. It contains the following buttons for selecting the mode for defining points:

  P (***Position***) - define the point position by clicking with the mouse in the Drawing Area.

  C (Constrain) - constrain to a control point selected with the mouse.

  U (Use) - use the position of a control point selected with the mouse.

  V (Value) - specify X, Y and Z coordinates of the point using the keyboard.

  The buttons become active for any operation that requires the user to define one or more points.
- The ***Hierarchy and Focus Controls*** navigate through composite objects such as groups and viewports in order to edit their subobjects.
- The ***Flip and Transform Controls*** may be used to flip, rotate or scale the object by a precise value.
- The ***Zoom and Rotate Controls*** let you control your view of the drawing area.

The ***Status Panel*** contains two rows of controls:

- The top row displays the name and type of the selected object; it also contains controls for quick access to the geometrical dynamics, actions, custom data and aliases attached to the object, if any. On the right of the top row there is a control that displays the cursor position in both the screen and world coordinates.
- The bottom row of the *Status Area* contains the *Prompt Area* that displays messages and prompts for action.

Stopping the GLG Graphics Builder

Use *File*, *Exit* to close the Builder; see page 301.

Creating a Viewport

A viewport is a GLG object that represents a cross-platform window used as a backdrop and container for other objects. The viewport also defines what part of the GLG drawing's infinite coordinate space will be visible on the screen. Although the Builder allows you to draw objects in its Drawing Area without a viewport, you must use a viewport as a container for a drawing if you want to display the drawing in a program.

When the Builder first starts, it creates a default viewport, names it "$Widget" and brings editing focus into the viewport, which is equivalent to the *File, New Widget* menu option. You'll see this viewport when you save the drawing.

When you start a new drawing using the *File, New* menu option, we recommend that you draw a viewport first, to act as a container for other objects. To draw the viewport:

1. Use the *Object*, *Create*, *Viewport* menu option, or click on the *Viewport* button  in the object palette in the upper left of the Builder window, then click on two points in the drawing to define the viewport's corners.

To draw objects inside the viewport, we need to "open" it to get inside:

2. If the viewport is not selected, select it by clicking on it with the mouse.

3. Use the *Traverse*, *Hierarchy Down* menu option, or click on the down arrow  in the hierarchy controls in the lower left of the Builder window.

Once you have created and opened a viewport, you can draw objects within its boundaries. To use the viewport in program, it must be named "$Widget".

Saving a Drawing

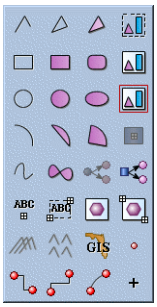
As you work on a drawing, use *File*, *Save* or *Save As* to save the drawing to a file; see page 298.

If you were editing objects inside the viewport, saving the drawing will bring you back to the top level of hierarchy. To return back to editing objects in the viewport, select the viewport and click on the *Hierarchy Down* button.

You should save your drawing frequently, so you can back out changes by reverting to the last saved version of the file. The Builder also provides an explicit Undo function, but some of the advanced operations, such as exploding or changing constraints, can not be undone.

Drawing an Object

Once you have created and opened a viewport, you can draw other objects within it.



Object Palette

To draw most GLG objects, you use the options on the *Object*, *Create* submenu, or pick a shape from the *Object Palette* in the upper left of the Builder window. The buttons in the *Object Palette* have tooltips that provide information about each button's function.

Not all the buttons on the palette correspond to distinct GLG object types. Some of the buttons are shortcuts to producing an object with particular attribute settings, provided for convenience. For example, both the *Arc* and *Circle* buttons create an arc object, but the angle of it is preset to 360 degrees when the *Circle* button is used.

The *Status Panel* at the bottom of the Builder window prompts you to specify the geometry and/or parameters of the new object. By default, you specify geometry by clicking on points inside the drawing area. Alternatively, you can change the way you specify points by using the *Point Controls* P C U V Position in the lower left of the window.

- *P* stands for *Position*. Click on a spot to specify the position of the new point. This method is the default.
- *C* stands for *Constrain.* Click on an existing control point to constrain the new point to the existing control point; see page 235.
- *U* stands for *Use Position.* Click on an existing control point to use the same coordinate values for the new point. The points merely use the same coordinates; they are not constrained, so moving one point has no effect on the other.
- *V* stands for *Value*. The Builder prompts you to specify the position of the new point by typing values in a dialog.

For most objects, you only need to specify a few points. The Builder always prompts you for all the information needed to create a particular object: the prompt is displayed at the bottom of the drawing area. For help in creating a particular shape, see the *Create* chapter on page 326.

For example, to draw a circle, you can click on the button. The Builder prompts you to specify the circle's center and another point that defines the radius.

To align the points of an object, use the options on the *Options*, *Snap To* submenu; see page 356. When you select a *Snap To* value, specifying coordinates with the mouse becomes less precise, because the point values are rounded off. *Snap To* only affects mouse selection.

GLG Objects

Although the menus and buttons in the GLG Graphics Builder show a wide variety of different drawing possibilities, the underlying graphical types are more restricted in number. Many of the buttons are provided for convenience, and do not represent separate object types. The available object types are:

- The text object, which presents string data.
- Simple graphical objects, which are just shapes, such as polygons, parallelograms, arcs, and markers.
- Advanced objects, which are specialized arrangements of objects that provide special behaviors. The advanced objects are viewport, group, reference, series, square series, polyline, polysurface, connector and frame objects; see page 265.

Complete descriptions of all the GLG objects appear in the *GLG Objects* chapter on page 65.

Selecting an Object

The simplest way to select an object is to click on it with the mouse. When you select an object, its control and resize points appear.

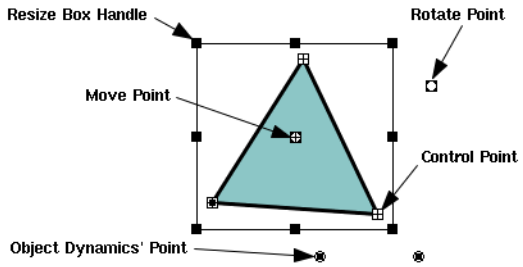
The **move point** appears at the object's center, it's a dynamically calculated point, provided for convenience in the Builder only. You can reposition an object precisely by *Shift*+clicking on the move point, and using the arrows in the Object Move Point dialog. To avoid accidental movement while 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).

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; see page 233.

If an object has geometrical dynamics attached, the control points of the dynamics will also be displayed. The **object dynamics' points** may be positioned the same way as other control points.



Object Selection: Control Points

The *Options, Selection Options, Selection Display* menu (*Ctrl-N* accelerator) can be used to change the selection display to show just the resize box, just the control points or both for convenience. When editing large groups with a lot of control points, set the selection display to show just the resize box to speed up group selection.

The *Options, Selection Options, Control Points Display* menu can be used to enable or disable display of control points of the object's dynamics.

To select an object with no fill, click on its edge. The *FillType* attribute controls this aspect of an object's appearance.

To select objects that are located close to each other, use *Shift+click* to bring 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.

The move point, rotate point and the resize box's points are provided for convenience in the Builder only. The control points, on the other hand, are real object points and may be accessed as object attributes or (if named) resources. For objects with a fixed number of control points, the points may be also accessed using the point's default resource name: *Point1*, *Point2*, *Point3*, etc. For objects with a variable number of control points, such as a polygon and its subclasses, the points can be accessed at run time by using the *GlgGetElement* function or method without naming the points in the Builder.

In the Windows environment, some viewports require special treatment to move. For instructions on dealing with this special case, see page 333.

Multiple Selection

To select more than one object, click and drag the mouse in the drawing to define a rectangular area: all objects that intersect this area will be selected. A temporary group is created to hold all selected objects; the group will be discarded when the objects are unselected. The temporary group is named "$TempGroup", and this name is displayed in the *Status Panel* when temporary group is created.

To add or delete objects from the selection, *Ctrl*-click on the objects with the left mouse button. For example, you could select multiple objects by *Ctrl*-clicking on them with the mouse. *Ctrl*-clicking on an object which is already selected deletes it from the selection.

The *Edit* menu provides more selection options: *Select All*, *Select Multiple Objects* and *Select Rectangular Area*, which may be used when the drawing has no empty space, making it impossible to define the selection rectangle with the mouse without selecting an object.

The *Arrange*, *Permanent Group* option changes the group type from temporary to permanent and back. The *Arrange* menu also provides explicit options for creating both temporary and permanent groups, as well as selecting multiple objects.

A permanent group can also be created by using *Object*, *Create*, *Group*, and drawing a rectangle that touches or encloses all the objects you want to include in the group. Release the objects from the group by selecting the group and using *Arrange*, *Explode*, *Object*. See the *Associating Objects Together* chapter on page 266 for information on how to perform an action on objects in a group.

Editing Objects

Creating an object adds a graphical object to the drawing and another branch to the object hierarchy. The object is created with default attributes that control its appearance. The Builder lets you change the object. You can:

- To change its appearance, edit its attributes.
- To change its size, move its box points.
- To change its geometry, move or edit its control points.
- To flip the object, click on one of the *Flip Object* icons in the *Control Panel* on the left of the Drawing Area.
- To rotate the object, move its rotate point.
- To scale or rotate the object by a precise amount, click on the Transform Object icon in the *Control Panel* on the left of the Drawing Area, then select a desired transformation type and define its parameters.
- Constrain the object (or any of its attributes) to another object or attribute; see page 235.
- Define and attach dynamic transformations to the object or to its attributes; see page 238.
- For gradient fill, cast shadows, arrowheads and fill dynamics, attach *Rendering* to the selected object; see page 271.
- Attach *Box Attributes* to the selected text object; see page 271.
- Attach *Custom Properties*, *Aliases* and *History* objects to the selected object; see page 273, page 274 and page 272.

To prototype the object's run-time behavior, you can animate it with simulated or random data; see page 256.

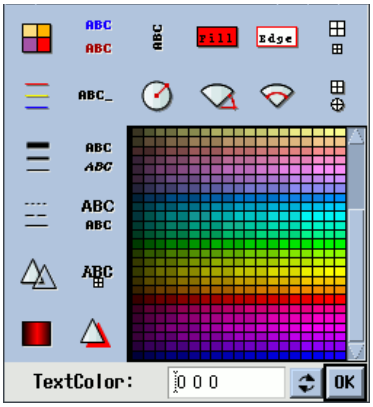
Editing Attributes

The attributes of an object are the characteristics that control its appearance. Each object type has its own set of attributes that are used to draw the object. Because the GLG objects are arranged in a hierarchy, an attribute is usually an object with its own attributes, constraints, and transformations. In the Builder, you can edit not only the attributes of a graphical object, but also the attributes of its attributes.

For a complete discussion of the attributes of the GLG objects, see the *Structure of a GLG Drawing* chapter.

Edit Toolbox

The *Edit Toolbox* provides a fast access to editing attributes of an object or a group of objects, and may be activated with either the *Object, Edit Toolbox*, or the *Edit Toolbox* toolbar button. The toolbox also provides a direct, single-click access to common rendering and text box attributes (i.e. gradient and shadow colors, text box color and line attributes), which otherwise require several mouse clicks to be accessed. When activated, it displays the most common graphical attributes of the selected object (or group of selected objects) and provides menus and palettes for point-and-click attribute editing. It also displays the name of the selected attribute and provides a text entry box for entering its value directly, in addition to the palettes and menus. To apply a new attribute value entered in the text entry box, press either the *Enter* key or *Apply* button. This picture shows the Edit Toolbox with a color palette for editing *TextColor* attribute:



The buttons in the toolbox have tooltips that show names of attributes associated with buttons' icons. Only the attribute buttons that are applicable to the currently selected object or group of objects are highlighted and active.

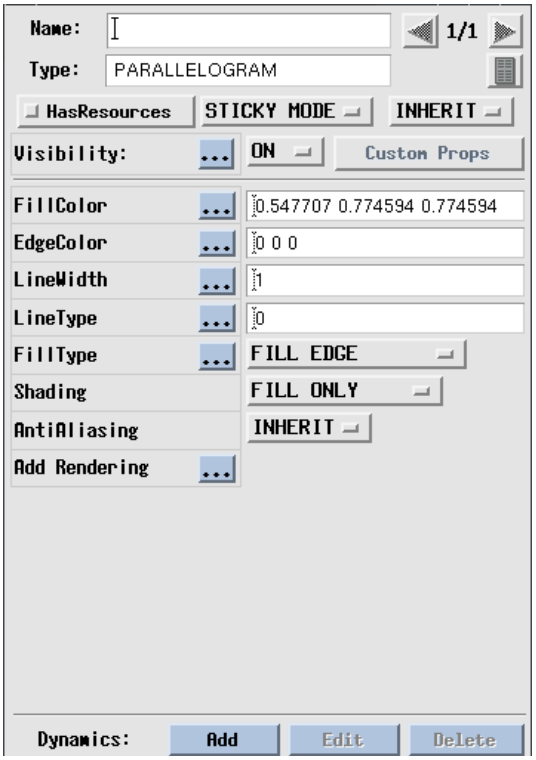
Properties Dialog

The properties dialog provides access to the object attributes for more specific editing. In addition to changing attribute values, it also enables assigning resource names to attribute objects, editing constraints and attaching attribute transformations.

To edit an attribute of a graphical object:

1. Select the object.

2. Use *Object*, *Properties* or click on the *Properties* button  on the toolbar to show the *Selected Object Properties* dialog. This dialog shows a list of attributes, with the current values. The list of attributes differs according to the object type. This example shows the properties of an unnamed parallelogram:



3. To edit an attribute, change its value.

The top panel of the properties dialog contains attributes common for all graphical objects, such as an object's *Name*, *Type*, *Visibility*, *HasResources*, etc. The middle part of the dialog contains object attributes that vary depending on the object type. At the bottom of the dialog there are buttons for adding and editing geometrical transformations, such as move, scale or rotate.

The *Properties* dialog of special objects, such as rendering attributes, text box attributes, font tables and viewport light attributes objects, also contain the *Mark* button to facilitate an easy reuse of these objects. The *Add or Use Marked Object* option of the *Edit* menu described on page 309 may be used to reuse these objects. If an object has custom properties attached, the *Custom Properties* button at the top of the dialog provides access to the custom property editing dialog.

While a value of an object attribute may be edited right in the *Properties* dialog, most attributes are objects themselves and have other properties in addition to the value. For attributes that are objects, an ellipsis button ... lets you open a separate *Attribute* dialog for editing the attribute value and other attribute properties.

Attribute objects may also have transformations, alarms and tags attached, in which case they are annotated with *"X", "A"* or *"T"* buttons on the right side of the *Properties* dialog. These buttons may also be used as shortcuts for accessing the dynamics, alarms or tags attached to an object's attributes.

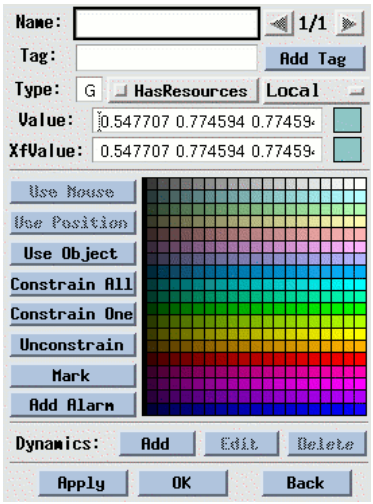
If an attribute has a transformation attached that overrides the value of the attribute, the value of the attribute in the *Properties* dialog will be read-only.

Attribute Dialog

The content of the attribute dialogs differ according to the attribute and its data type (string, scalar, or geometric). However, all the dialogs include:

- Text boxes for entering attribute name and value.
- Text fields showing an attribute's data type (D, S or G) and transformed value (*XfValue*).
- A way to change the attribute value via a palette, a menu with list of possible values or a spinner.
- Toggles for setting the attribute's *HasResources* and *Global* attributes.
- A button for adding or editing a tag and a text field showing tag information.
- Buttons for reusing attributes of other objects in the drawing.
- A button for marking the attribute for reuse; see page 259.
- Buttons for constraining the attribute; see page 236.
- Buttons for adding, editing, or deleting transformations for the attribute; see page 241.
- A button for adding or editing an alarm to monitor the attribute's value.

The following picture shows an *Attribute* dialog for a color attribute.



For color attributes, the colored boxes on the right side of the *Value* and *XfValue* fields show both the color and transformed color of the attribute.

The *Options, Color Options, 255 Color Display* option may be checked to display color RGB value in the range 0-255. By default, RGB values use the range from 0 to 1. The *ColorDisplay255* parameter in the *glg_config* file may be set to 1 to permanently use the 255 color range.

In addition to a color palette shown in the picture, the Builder provides a custom color palette. To switch color palettes, use *Options, Color Options, Swap Color Palettes*, or simply *Ctrl+click* on the color palette. Refer to the OEM Customization chapter for information on how to supply your own custom color palette.

You can also *Shift-click* in the color palette or select *Options, Color Options, Pastel Colors* to switch between pastel and regular colors.

If an attribute object has transformations, alarm or tag attached, it will be annotated with *"X", "A"* or *"T"* buttons on the right side of the attribute row in the object's *Properties* dialog. These buttons may also be used as shortcuts for accessing the dynamics, alarms or tags attached to an object's attributes, bypassing the *Attribute* dialog.

XfValue is an edit-only text field that shows the transformed value of the attribute. If the attribute does not have any transformation attached, the transformed value will be the same as the attribute value, otherwise it will show the transformed value of the attribute that reflects the combined effect of all transformations attached to the attribute.

For convenience, all object attributes have default attributes names. Therefore, you can edit an attribute of an object by browsing its resource hierarchy and locating the attribute; see page 245. For a complete list of the objects' attributes and their default attribute names, see the *Appendix C: GLG Object's Attribute Table* chapter on page 369 of the *GLG Programming Reference Manual*.

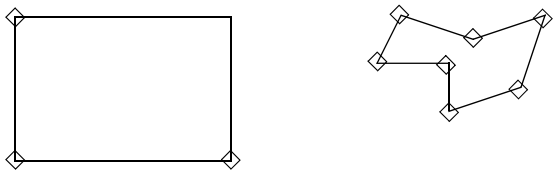
If an attribute has a transformation attached that overrides the attribute's value, the Value field will be read-only.

If the *Edit Dynamics* or *Edit Alarm* dialogs are open to edit a transformation or alarm attached to the attribute object, the *Attribute* dialog is disabled for the duration of the editing. This behavior may be changed using the *ModalXformDialogs* parameter of the glg_config file.

Editing Control Points

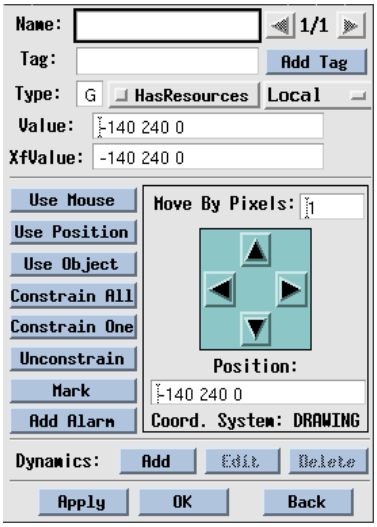
A control point is just a data attribute of an object, containing the coordinates of one point. However, since the Builder uses a different technique for editing control points, they require extra attention.

A minimal number of control points defines the basic geometry of each object. Depending on the shape, additional points may be calculated dynamically. For example, three control points define a parallelogram, and the last vertex is calculated dynamically. However, a free-form polygon has a control point at each vertex.



A parallelogram and a polygon, with their control points

To see the attributes of a control point, *Shift+click* over the point. The *Control Point* dialog shows the attributes of the point, with arrows for precise movement of the point and buttons for manipulating the point.



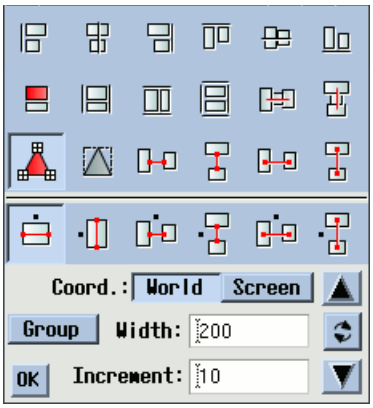
- *Use Mouse* positions the point where you click in the drawing area.
- *Use Position* positions the point at the same place as another control point that you click on. Resources or marked objects may also be used to define another control point.
- *Use Object* gives the point the same coordinate values as another control point that you click on. Resources or marked objects may also be used to define another control point.
- *Constrain One* replaces the point's existing constraints with a constraint to the point you click on; see page 235. Resources or marked objects may also be used to define the control point to constrain to. (*Constrain One* is enabled only if activated in the *glg_config* file.)
- *Constrain All* is similar to *Constrain One,* except that it constrains not only the point itself, but also all points constrained to it; see page 235.

- *Unconstrain* removes all constraints from the point as well as all other points constrained to it.
- *Mark* stores the point on the Builder's clipboard. You can mark up to five objects; see page 259.
- *Add or Edit Alarm* adds a new alarm object to monitor the points' value or edits an existing alarm.
- *Add Dynamics*, *Edit Dynamics*, and *Delete Dynamics* manipulate transformations of the control point; see page 241.
- The directional arrows may be used to move the point precisely by one or more screen pixels as defined by the *Move By Pixels* field. A fractional amount such as 0.5 may be used for even more precise positioning, which might be required in the OpenGL version of the Builder.
- The *Position* field may be used to position the control point by specifying its coordinates in the currently selected coordinate system. The current coordinate system may be changed by the *Coordinate System* option of the *View* menu.

If more than one control point is located at the same spot, the Builder prompts you to select the one to edit by activating the arrow buttons in upper left corner of the *Control Point* dialog when the point is selected with the *Shift+click*.

Object Layout and Alignment

The *Layout Toolbox* provides point and click access to the align and layout features, and can be activated using *Layout*, *Layout Toolbox*, or pressing the *Layout Toolbox* button in the toolbar. The toolbox is split into two areas: the top panel contains icons for operations that does not require any parameters (*Align Top, Set Same Width, Distribute Evenly Across*, etc.) , while the bottom panel contains icons and controls for operations that requires a parameter, such as *Set Width* or *Set Horizontal Distance*. The toolbox's buttons have tooltips that show actions associated with buttons' icons. The following picture shows the *Layout Toolbox*:



The icons in the top panel of the *Layout Toolbox* operate on several objects and are active only when multiple objects or a group is selected. The first button in the second row, the *Select Anchor Object* button, has a bright red color and may be used to select the anchor object: the rest of the objects will be aligned relatively to the anchor object's extents. To select an anchor for several selected objects, click on the button, then click on one of the selected objects. When the anchor object is selected, the color of the icon turns green.

The first two buttons in the lower row of the area are highlighted with red color as well and can be used to switch between two layout modes: the *control points* and *bounding box mode*. In the *control points mode*, the object's control points are aligned. In the *bounding box mode*, the bounding box of objects is used for alignment. The difference can easily be seen by trying to align text objects with just one control point but different text extents. There are situations, though, where either one or the other alignment mode may be useful.

When an object is selected, the bottom panel of the *Layout Toolbox* may be used to display its width or height in either the world or screen coordinates by clicking on the *Set Width* or *Set Height* buttons in the bottom panel. When an anchor is selected, clicking on the buttons displays the width and height of the selected anchor object. For example, selecting a group object with the *Set Width* button active will display the width of the whole group. Clicking on the *Set Anchor Object* button and selecting an object inside the group as an anchor will display the width of the anchor.

The bottom panel of the toolbox contain a row of icons for operations that need a numerical parameter. For example, setting the width of an object requires the width parameter. These operations may be applied to a single object or to a group of objects. When a group is selected, the *Group Editing Mode* button on the left of the panel defines if the layout operation, such as *Set Width* or *Set Height*, will be applied to the group itself or to the objects the group contains.

The text entry box is provided for entering the required parameter, which is initialized to the corresponding parameter of the selected object, or the anchor object if it is defined. The *World* and *Screen* buttons above the text box control whether the value of the layout parameter, such as width, height or space, is interpreted as screen pixels or world coordinates. The *Increase* and *Decrease Value* buttons with arrows provide a convenient way to increase or decrease a desired parameter in steps, and with the provided increment. To apply a new parameter value entered in the text edit box, press either *Enter* key or *Apply*  button.

Some operations from the bottom panel may not be applicable to certain object types. For example, trying to set width or height of a text object with one control point would generate a corresponding warning message. The *Layout* pull-down menu also provides options for accessing align and layout operations via the menu.

Edit, Undo may be used to reverse erroneous layout or alignment actions.

Creating Constraints

A constraint causes one attribute to change along with another attribute.

You can constrain the same attribute for two different objects. For example, constraining the *FillColor* attribute of a yellow polygon to the *FillColor* attribute of a green circle turns the polygon green. In the future, changing the *FillColor* of either object will affect both of them.

You can also constrain any attributes that have the same data type (string, scalar, or geometric); for example, a label and incoming data, the radius of a circle to the number of sides it has, or a color and a control point.

In the object hierarchy, a constraint is a point where two attributes merge. The constraint is not a link between the values for an attribute; it actually merges the attribute values. When two attributes are constrained, one attribute value is replaced.

Constraining Similar Attributes

To create a constraint, you select the attribute that is to lose its value, and then constrain it to the attribute that replaces it. If the attribute is already constrained, the existing constraint is replaced by the new one unless you merge the constraints; see below.

To constrain an attribute to the same attribute in another object:

1. Select the object that has the attribute you want to constrain.

2. Use *Object*, *Properties* or click on the *Properties* button  to show the *Selected Object Properties* dialog. This dialog shows a list of attributes, with a  for attributes that can be edited in a dialog.

3. Click on the  to see the *Attribute* dialog.

4. Click on the *Constrain One* button in the *Attribute* dialog. To keep existing constraints, use the *Constrain All* button instead. (*Constrain One* is enabled only if activated in the *glg_config* file.)

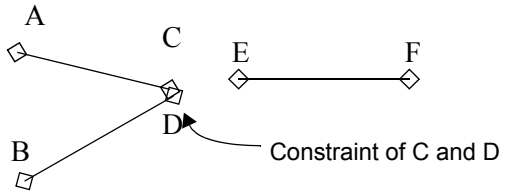
5. The Builder prompts you for the object to constrain to. To do so, click with the mouse on an object in the drawing.

With this method, the Builder automatically applies the constraint to the appropriate attribute. For example, selecting the *FillColor* attribute of a yellow polygon and then selecting a green circle to constrain to creates a constraint between the *FillColor* attributes of the two objects.

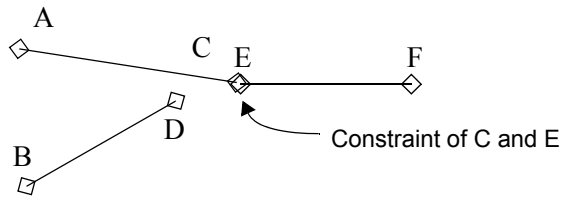
Merging Constraints

If you use the *Constrain One* button, any existing constraints are replaced by the one you create. However, you can add another constraint without removing the existing constraints; use the *Constrain All* button instead of the *Constrain One* button. (*Constrain One* is enabled only if activated in the *glg_config* file.)

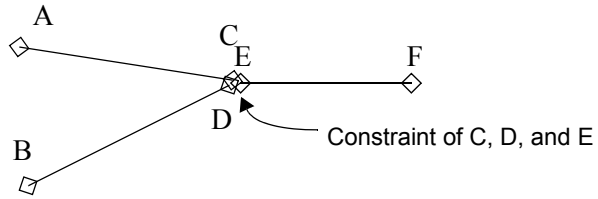
For example, you can constrain the points C and D.



If you then select C and use the *Constrain One* button to constrain it to E, the constraint between C and D is lost.



However, if you select C and use the *Constrain All* button instead of the *Constrain One* button, the constraint between C and D is preserved.



Constraining Different Attributes

To constrain different attributes that have the same data type, you can use the Builder's marking facility or refer to a named resource. To use marking, you open and mark the attribute value that you want to reuse, and then use it in the attribute whose value you want to replace. Alternatively, if the attribute already appears in the resource hierarchy, you can just use its value without having to mark it.

As with constraints between similar attributes, constraining an attribute that is already constrained replaces the existing constraint, but you can add another constraint and preserve the existing ones by merging them.

To constrain two different attributes:

1. Open the *Selected Object Properties* dialog for the object.

2. Open the *Attribute* dialog for the attribute that you want to reuse.

3. Click on the *Mark* button, and select a mark number to assign to the attribute value. The attribute value is now marked.

4. Open the other object's *Selected Object Properties* dialog.

5. Open the *Attribute* dialog.

6. Click on either the *Constrain All* button or the *Constrain One* button in the *Attribute* dialog. The Builder prompts you to select either a resource or a marked value. (*Constrain One* is enabled only if activated in the *glg_config* file.)

7. Click on *Use Marked* and select the mark number you assigned earlier (if only one mark was assigned, it'll be used automatically). To use a named resource instead of a marked value, click on *Use Resource* and select the resource from the *Resource Browser* dialog.

The value of the attribute you marked is applied to the other attribute. For example, constraining the *FillColor* attribute of a yellow circle to a control point of a green polygon makes the circle change color when you move the control point.

Constraining Control Points

Control points are constrained just like other attributes. Use *Shift+click* to see the attributes for the control point, then use either the *Constrain All* or the *Constrain One* button and select the control point to constrain to. (*Constrain One* is enabled only if activated in the *glg_config* file.)

Constraints Tracing

To trace constraints for a given attribute, mark the attribute as *Mark0* in the Attribute or Resource Object dialog and activate tracing via the *Options, Selection Options, Trace Attribute Constraints for Mark0* menu option.

To find objects in the drawing that have attributes constrained to the attribute marked with *Mark0*, select each object with the mouse. If the selected object has attributes that are constrained to the marked attribute, the Status Panel fields that display the object's *Name* and *Type* will be highlighted in red. If the object has a geometric transformation, an action or custom data whose attributes are constrained to the marked attribute, the corresponding buttons in the Status Panel will also be highlighted. To narrow the search, follow the highlighted items and their properties.

The attributes constrained to the marked attribute will also be highlighted with a red outline in the property dialogs, such as Object Properties, Object Dynamics, Attribute Object and others. If the attribute itself is constrained, a solid outline is used. If an attribute itself is not constrained, but has a transformation whose parameters are constrained, a dashed outline is used.

Stars are used to annotate constrained items in list dialogs, such as Custom Properties or Object Dynamics List. If an item is constrained, it is annotated with two stars. If the item itself is not constrained, but has a transformation whose parameters are constrained, it is annotated with one star.

In drawings with a large number of objects, group multiple objects in a temporary group and check if *Name* and *Type* field in the Status Panel are highlighted.

Defining Transformations and Adding Dynamics

In addition to changing values of object attributes directly, dynamic behavior may be achieved by adding a dynamic transformation to an entire object or an object attribute. For example, a rotate transformation may be attached to an object to rotate it by changing a rotation angle, or a *Color List* transformation may be attached to an object's *FillColor* attribute to change the color depending on an input value.

A transformation is applied to change the value of an entity. It operates on the actual value of the object, producing an effective value which is used in rendering the drawing. The GLG Graphics Builder provides three sets of basic transformations, one set for each basic data type (string, scalar, and geometric). See the *GLG Objects* chapter for descriptions of the different transformation objects.

In the Builder, there are three different ways to apply a geometrical transformation to an object:

- *Transform Points* makes an immediate and permanent change to the definition of the object by changing coordinate values of its points, without creating a transformation object.
- A static (or matrix) transformation is attached to the object to change its appearance without changing coordinates of the object's control points. A static transformation doesn't change the object definition; instead, it "projects" the object so that it appears in a different place or in a different shape. When several static transformations are applied, they are merged into one static transformation with combined parameters. The transformation parameters cannot be edited.
- A dynamic (or parametric) transformation has parameters that can be edited or animated at runtime by providing input data to the transformation; see page 256. Since dynamic transformations are intended primarily for runtime animation, their parameters are set to initial values that do not change the object's initial position. When several dynamic transformations are attached to an object, each transformation can be animated independently of the others.

A transformation can be attached to an entire object, to an attribute of an object, to a control point, or to the entire drawing. If a composite object such as a group has a static or dynamic transformation added to it, the transformation transforms all objects inside the group. In other words, an object in a drawing inherits the combined effect of all transformations attached to it or its parents.

Adding Geometrical Dynamics and Transforming an Object

You can define a geometrical transformation by using a dialog to set the parameters of the transformation. When you transform an object, you can use one of three methods:

- **Transform object points** to make a permanent change to the object's definition.
- **Add a static transformation** to make a one-time, reversible change to the object's geometry. The transformation may be later deleted to restore the object's original appearance.
- **Add a dynamic transformation** to control an object's geometry or position using dynamic parameters which may be animated with input data.

Transforming Object's Points

To transform the points of an object, making a permanent change to their coordinates:

1. Select the object and use *Object*, *Transform Points*. Alternatively, click on the *Transform Object Points* toolbar icon or *Transform Object* button in the *Control Panel*. The *Transform Points* dialog appears.

2. Select the transformation type using the *Transformation Type* option. The content of the *Transform Points* dialog changes, depending on the transformation you select. You can also switch to creating a static or dynamic transformation using the *Action* option in the dialog.

3. Specify the values to use in the transformation by typing them in the boxes. Alternatively, use the buttons on the right to supply the values using the mouse; the Builder prompts you for each required value.

4. Click on the *Apply* button to change the definition of the object by applying the transformation to the object.

5. To undo a transformation right after you apply it, use the *Reverse* button to specify an inverse transformation, and click on *Apply* again.

Transforming the points changes the object definition permanently, but does not add a transformation object to the object hierarchy.

Creating a Transformation Object

To create a transformation object attached to a graphical object:

1. Select the object and use *Object*, *Add Static Transformation*, or *Add Dynamics*. Alternatively, open the *Selected Object Properties* dialog for the object, and click on the *Add Dynamics* button, or click on the *Add Dynamics* toolbar icon. A dialog for specifying the transformation appears; its title depends on the transformation you selected.

2. Select the transformation type using the *Transformation Type* option. The content of the dialog changes, depending on the transformation you select. You can also switch between transforming points and creating static or dynamic transformation using the *Action* option in the dialog.

3. Specify the values to use in the transformation by typing them in the boxes. Alternatively, use the buttons on the right to supply the values using the mouse; the Builder prompts you for each required value.

4. Optional step for **dynamic transformations**: enter a name for the transformation's controlling parameter into the *Variable Name* field, and define the ranges that control mapping of input data to the change of the *Factor* parameter. If the default ranges are modified, a *Range Conversion* transformation will be attached to the *Factor* parameter, and

the name entered in the *Variable Name* field will be assigned to the *Input Value* parameter of *Range Conversion*. If the default ranges are not modified, the name will be assigned to the *Factor* attribute of the attached dynamics.

5. Click on the *Apply* button to change the definition of the object by applying the transformation to the object.

6. To undo a static transformation right after you apply it, use the *Reverse* button to specify an inverse transformation, and click on *Apply* again. To undo a dynamic transformation, delete the attached transformation using *Object, Delete Dynamics* or the *Delete Dynamics* toolbar icon.

Defining a static or dynamic transformation adds it to the transformation list, so you can access it using *Object*, *Edit Dynamics*. If you want to attach the same transformation definition to other objects, you can use the *Mark Object* or *Mark List* buttons in the *Edit Dynamics* list; see page 261.

Several dynamic transformations can be attached to an object to move, scale and rotate it depending on several dynamic parameters. The order of transformations is important: if the order is changed, the result of applying several transformations to an object will be different. The *Edit Dynamics* dialog provides controls for changing the order of transformations in the list.

The MoveMode Attribute

The *MoveMode* attribute is used to preserve a relative position of centers of rotation and scale dynamics attached to the object from changing when the object is moved with the mouse. If the *MoveMove* is set to STICKY MODE, the center of rotate or scale transformation will be moved together with the object. If it is set to MOVE POINTS, the center of the transformation will not be moved with the object.

For example, an object rotating around its center with *MoveMode* set to STICKY MODE will still rotate around its center after being moved. With *MoveMode* set to MOVE POINTS, the center of rotation will not move along with the object, and the object will rotate around the old center position even after the object has been moved.

If the *MoveMode* is set to MOVE BY XFORM, moving or transforming the object in some other way results in adding a static xform to the object, instead of changing the coordinates of the object's control points. The added transformation equally transforms the object's control points and the attached transformation's center points, preserving their relative position. This may be used when you want to preserve the original coordinates of an object's control points.

The *MoveMode* attribute is located in the *Selected Object Properties* dialog next to the *HasResources* flag.

Adding Attribute Dynamics

Attribute dynamics are accomplished by adding a dynamic transformation to an attribute object. Examples of commonly used attribute dynamics include visibility, color and text dynamics.

The data type of the attribute controls the type of transformation you can apply to it. For geometric values that represent points in the drawing, the dialogs are similar to those for transforming objects. For other attribute objects that represent scalar, string or color values, the transformation type is selected from a list displayed inside the *Attribute* dialog.

The Builder provides various types of dynamic transformations that can be used as building blocks for implementing elaborate dynamic behavior. In addition to the stock transformation types, the Builder also supplies easy to use predefined dynamics options for implementing the most common dynamic actions.

Custom predefined dynamics may be defined using the Enterprise version of the Builder started with the *-oem* command-line option, see the *Custom Predefined Dynamics* section on page 284. Custom predefined dynamics may be used by the system integrators to extend GLG editors with elaborate application-specific dynamics.

To define a transformation object attached to an object's attribute:

1. Select the graphical object that has the attribute you want to transform.

2. Use *Object*, *Properties* or click on the *Properties* button  to show the *Selected Object Properties* dialog. This dialog shows a list of attributes, with a for attributes that can be edited in a further dialog.

3. Click on the  for an attribute, to see the *Attribute* dialog.

4. Click on the *Add Dynamics* button in the *Attribute* dialog. A list of transformation types appears. The content of the transformation list depends on the data type of the attribute. The predefined dynamics are listed first, and the *Show More* button may be used to show all available stock transformation types. See the *Transformation Object* chapter on page 151 for specifications of the transformations.

5. Click on the transformation name to add the new transformation to the attribute, then edit the transformation's parameters in the activated *Edit Dynamics* dialog.

Once the transformation has been defined, you can edit it using the *Edit Dynamics* button in the *Attribute* dialog. You can attach the same transformation definition to other attributes by using the *Mark Object* and *Mark List* buttons in the *Edit Dynamics* dialog; see page 261.

Only one transformation can be added to an attribute (except for control points). To create complex attribute dynamics that depend on multiple parameters, dynamic transformations may be "chained" by attaching additional transformations to the attributes of the first transformation. When such recursive transformations are edited, the title of the *Edit Dynamics* dialog reflects the current nesting level of the transformation.

Adding Dynamics to Control Points

Control points are special attributes of an object that define its geometry. A geometrical transformation can be added to a control point to move, scale or rotate the point based on a value of a dynamic parameter.

To add a dynamic transformation to a control point, *Shift+click* on the point to show the *Control Point* dialog, click on the *Add Dynamics* button, then follow steps 2-5 described in the *Creating a Transformation Object* section on to add a transformation.

The *Add Dynamics* button in the *Control Point* dialog is activated only for real control points. It is disabled for the object move point and resize points, which are displayed only in the Builder for convenience of editing .

Several dynamic transformations can be attached to a point to move, scale and rotate it depending on several dynamic parameters. The order of transformations is important: if the order is changed, the resulting point position will be different. The *Edit Dynamics* dialog provides controls for changing the order of transformations in the list.

Editing Transformations

For each object or attribute with transformations, the Builder maintains a list of attached transformations. The list shows both the static and dynamic transformations, but you can only edit parameters of the dynamic transformations.

To edit transformations:

1. First, display the transformation list:

   a. For a graphical object, use *Object*, *Edit Dynamics* to see a list of transformations. Alternatively, click on the *Edit Dynamics* toolbar icon or click on the *Xform* button in the *Status Panel* below the drawing area. If the *Properties* dialog is open, you can also use the *Edit Dynamics* button at the bottom of the dialog.

   b. For an attribute, use *Object*, *Properties* to show the *Selected Object Properties* dialog, and click on the … to see the *Attribute* dialog. Finally, click on the *Edit Dynamics* button. Alternatively, click on the "*X*" button on the right side of the attribute row in the *Properties* dialog.

   c. For a control point, use *Shift+click* to see the *Control Point* dialog, and click on the *Edit Dynamics* button.

   A list of transformations is displayed on the left side of the *Edit Dynamics* dialog.

2. In the *Edit Dynamics* dialog, select a transformation from a list on the left side of the dialog. The transformation attributes appear on the right side of the dialog.

The transformations are listed in the order you created them, with the first transformation at the bottom of the list. The new transformations are added at the top of the list. If the coordinate system is set to the *Object Coordinate System*, a new transformation is added at the bottom of the list, see page 264.

The *Up* and *Down* buttons on the right side of the *Transformation List* may be used to reorder transformations by moving the selected transformation up or down. The order of transformations is important: reordering transformations changes the way the object is transformed. The *Delete* button may be used to remove the selected transformation.

If you add several static (matrix) transformations consecutively, they are merged into a single matrix transformation.

Deleting Transformations

The *Delete Dynamics* toolbar icon can be used to delete the transformation that was added last (displayed at the top of the list). The Builder does not confirm the deletion of a transformation. If the coordinate system is set to the *Object Coordinate System, Delete Dynamics* deletes the transformation which was added first (displayed at the bottom of the list), see page 264.

The *Delete* button of the *Edit Dynamics* dialog may be used to delete the currently selected transformation from any position in the list.

To remove the first transformation:

- For an object, select it and use the *Object*, *Delete Dynamics* menu option or click on the *Delete Dynamics* toolbar icon. If the *Properties* dialog is open, you can also use the *Delete Dynamics* button at the bottom of the dialog.
- For an attribute, select the object, use *Object*, *Properties* to show the *Selected Object Properties* dialog, and click on the ... to see the *Attribute* dialog. Finally, click on the *Delete Dynamics* button.

  Alternatively, click on the *"X"* button on the right side of the attribute row in the *Properties* dialog, then click on the *Delete* button in the *Edit Dynamics* dialog.
- For a control point, use *Shift+click* to see the *Control Point* dialog, and click on the *Delete Dynamics* button.

To remove a selected transformation from a list of transformations:

- Display a list of transformations as described in the *Editing Transformations* chapter.
- Select the transformation in the list to delete.
- Click on the *Delete* button at the bottom of the *Transformation List* dialog.

If you added several matrix transformations consecutively, they are merged into a single matrix transformation and are all deleted together.

Traversing Transformed Objects (advanced)

For an object with at least one transformation attached to it, the Builder lets you view the untransformed object. The Builder's main window shows the transformed object, and you use *Traverse*, *Transformation Down* and *Traverse*, *Up* to move between the transformed and untransformed view of the object.

Using View and Screen Transformations of the Viewport (advanced)

A *Viewport* uses two sets of transformations: the "outside" transformations are used to position the viewport itself, and the "inside" transformations are used to draw the objects inside the viewport. When the viewport is selected, the *Add, Edit* and *Delete Dynamics* buttons of the *Properties* dialog modify an outside transformation. If the editing focus is inside the viewport with no objects selected, the buttons modify the inside transformations which may be used to zoom, pan or rotate all objects in a viewport without creating a group to hold them. After either the inside or outside transformation is added, it may be edited with the *Edit Dynamics* button.

When a list of inner transformations of the viewport is displayed, it also shows the viewport's zoom transformation. This is a matrix transformation which is automatically created and used when changing the view in the Builder. It is always at the bottom of the list and cannot be deleted.

The viewport's screen object also has a special drawing transformation used to adjust the rendering of the viewport when the viewport's size changes. Clicking on the *More* button in the *Viewport Properties* dialog and then on the *Edit Dynamic* button in the *Screen Properties* dialog provides access to the parameters of this transformation. The X and Y scaling factors of the transformation may be used to create fixed screen offsets as described in the the *Screen* section of the *Simple Graphical Objects* chapter.

Using Resources

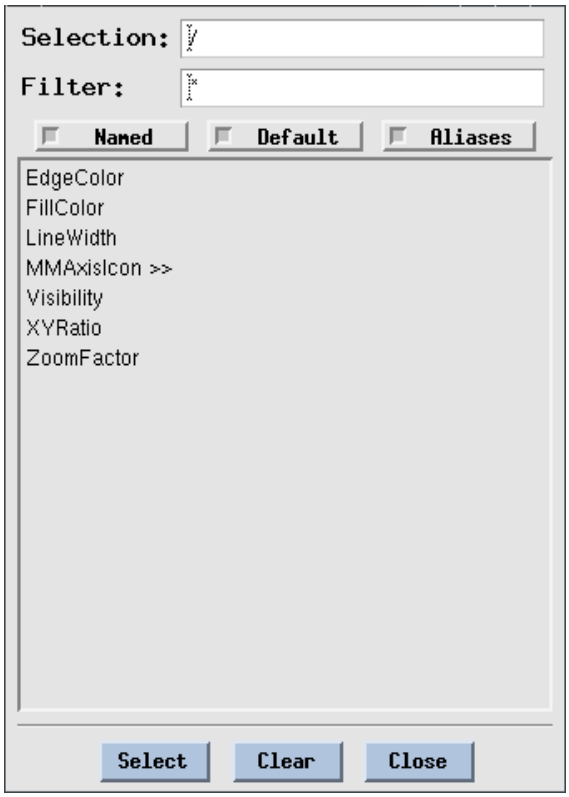
A resource is an object or attribute that is accessible by name. The Builder includes a browser which shows the structure of the named resources in a drawing — its resource hierarchy. The resource hierarchy is not the same as the object hierarchy; it contains only those objects that you have named and defined as having or being resources. See the *Structure of a GLG Drawing* chapter for more information.

Giving an object a name adds it to the resource hierarchy. The object's position in the hierarchy is controlled by the *HasResources* flags of its parent objects. If a parent object has the *HasResources* flag set to *YES*, the object is added to the hierarchy below the parent. If the object has no ancestors in the hierarchy, it appears at the highest level. The *HasResources* flag defines hierarchy levels similar to directories of a file system.

When naming an object, it's also convenient to consider the settings of its *HasResources* flag at the same time.

To see resources of an object, select it and use either *Object*, *Resources* menu option or the *Resources* toolbar button to bring the *Resource Browser* dialog, see page 341. To see resources of the whole drawing, unselect the object by pressing the *Esc* key. Selecting another object while the Resource Browser is active will show its resources in the Resource Browser.

The *Resource Browser* dialog shows the resources organized like a file and directory structure, with levels of hierarchy corresponding to the *HasResources*=YES settings.



Composite resources that contain other resources are annotated with the >> suffix added to the resource name. You can navigate between the hierarchy layers by double-clicking on the entries with the >> symbol at the end.

>> following a resource name means that it is either a geometrical object (i.e. polygon, arc, etc.) with a set of default attributes, or it has its *HasResources* flag set to YES and may contain other named resources. In either case, double clicking on such a composite resource opens another level of hierarchy, showing the resources inside of it.

Entries with no >> symbol are usually attributes of objects. Clicking on such entries selects them, showing the *Resource Object* dialog for editing that attribute.

The selection box at the top of the Resource Browser shows the resource path of the currently selected resource, using / to separate hierarchy levels. This notation lets you specify a "path" to a resource, just as you would specify a path to a file. When the Resource Browser is used in the Builder, the path may start with one of the following symbols:

/ - top level (resources of the whole drawing area).

. - the selected object resources.

~ - resources of the viewport with the current editing focus (as a result of the *Set Focus* button).

$config - global configuration resources of the Builder.

The first three symbols may also be displayed as entries in the Resource Browser to let you select a subset of resources for browsing. These entries are available only in the Resource Browser and not in the API.

The Resource Browser also contains the ".." entry representing the previous level of the hierarchy relatively to the currently selected resource.

The resource browser provides three toggles which can be used to control which resources are displayed in the browser: *named resources*, *default resources*, *aliases* or any combination or them. By default, all three resource categories are displayed. Commonly used attributes of most objects may be accessed via their *default resource names* such as *FillColor* or *LineWidth*, without requiring the user to name each resource. If an object attribute such as *FillColor* is named, it may be accessed by both the user-defined name and the default resource name.

The *Filter* field of the resource browser defines a regular expression to apply to the entries on one level of the hierarchy. Only the entries matching the filter expression on the current level of the resource hierarchy will be displayed. The * (any sequence of characters) and *?* (any character) wildcards may be used to construct the filter.

Guidelines for Naming Resources

Although the resource hierarchy is totally flexible and should be organized to meet your requirements, we suggest the following guidelines for including resources:

- Name the object if you plan to animate it. Animation requires access to a named resource.
- Name the object if it corresponds to a graphical object in the drawing that you want to access programmatically later on. This is not required, but it lets you use the resource hierarchy to locate any graphical object in the drawing, regardless of its visibility or location in the hierarchy.
- Name the object if you plan to edit its attributes frequently. This is not required, but it lets you use the resource hierarchy to locate and select the object's attributes.
- Name control points, if selecting them with the mouse would be difficult, or if you'll need to access the point programmatically later on. This is not required, but can simplify editing closely positioned points.
- It is not usually necessary to name transformations, though you should name the attribute of the transformation that you plan to animate.
- Do not assign the same name to different objects, and use care when cloning or copying objects, renaming the copies. Naming conflicts, where more than one object at the same level have the same name, can have unpredictable effects.

These guidelines are not intended to force you into naming more objects than you find useful, but they may help you to make better use of the resource hierarchy's ability to structure the drawing and provide access to objects, especially as you begin using the Builder.

Adding and Deleting Resources

Naming an object automatically adds it to the resource hierarchy; its location is controlled by the *HasResources* flag of its parent object. When you delete an object from the drawing, it is also deleted from the resource hierarchy.

Adding an Object to the Resource Hierarchy

To add an object to the resource hierarchy:

1. Select the object.
2. Use *Object*, *Properties* or use the *Properties* button  to see the attributes of the object.
3. Type a name for the object in the *Name* box.

The new name appears in the resource hierarchy.

A set of *default resource names* for the object's attributes are also added; they are automatically placed below the object, even if the *HasResources* flag is set to *NO*.

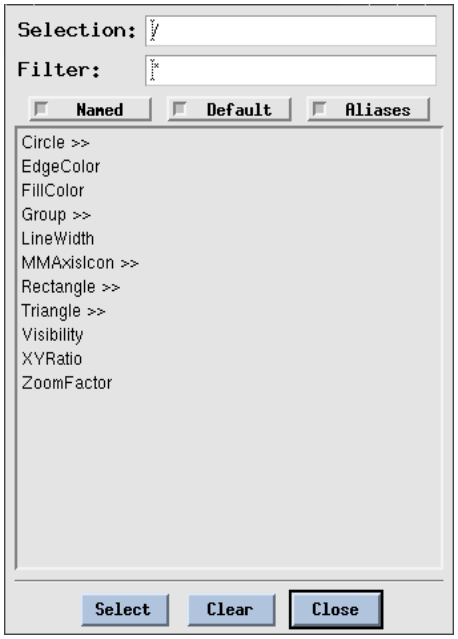
The object's attributes can also be named by selecting the attribute's ellipsis button  and entering a name into the *Name* box of the *Attribute* dialog. For named resources, both the user-defined name and default resource name will appear in the resource browser.

The default names will always appear as the resources of the object. The location of the user-defined names will be controlled by the object's *HasResources* flag. If the flag is set, named attributes will appear as resources of the object, otherwise they'll appear as resources of the closest parent with its *HasResources* set to YES.

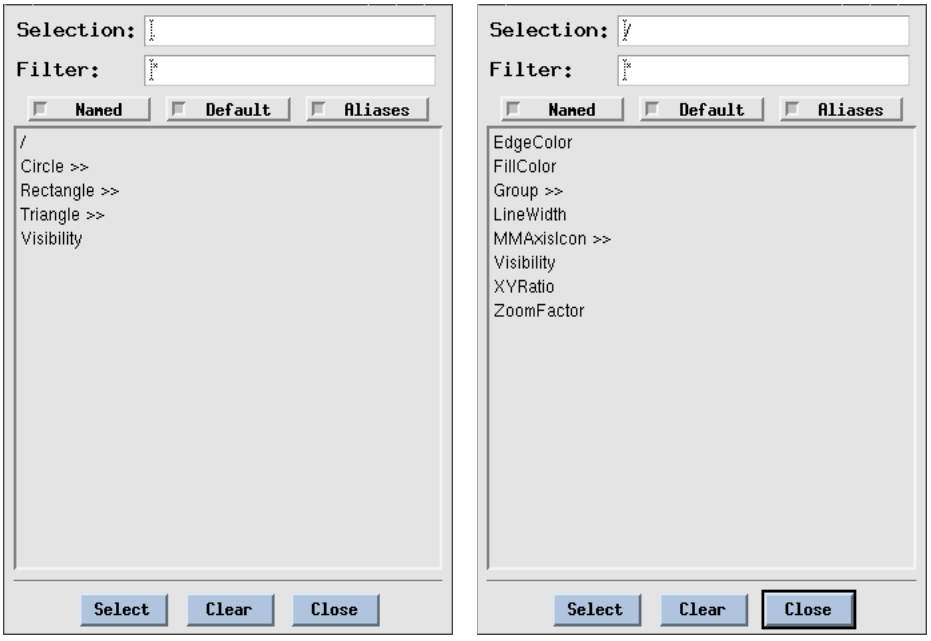
Defining the Hierarchy

Use the *HasResources* flag to control the organization of objects in the resource hierarchy.

For example, consider a group object named *Group*, with the members *Circle*, *Triangle*, and *Rectangle*. If the group object's *HasResources* flag is set to *NO*, the hierarchy is flat and the *Resource Browser* dialog shows all four objects at the same level.



Setting the group object's *HasResources* flag to *YES* defines an additional level of a hierarchy and makes the three objects into child objects of the group. The *Resource Browser* dialog shows *Group* as having child objects. Below, the picture on the left shows resources of the group, while the picture on the right shows resources of the Drawing Area.



Setting the *HasResources* flag also changes the path for locating an object. In this example, setting the group object's *HasResources* flag to *YES* changes the path to the *Circle* object from *$Widget/Circle* to *$Widget/Group/Circle*.

Deleting a Resource from the Hierarchy

To delete a named resource from the resource hierarchy, simply unname it. To remove the name from a named attribute object, follow the following steps:

1. Use *Object*, *Resources* or click on the *Resources* button  to see the resource hierarchy.

2. Locate the resource you want to delete, and click on it to select it.

3. Delete the characters from the *Name* box of the *Resource* dialog. The object will no longer appear in the resource hierarchy.

The above procedure will work only for named attributes. To remove the name of a geometrical object (i.e. polygon, arc, etc.), select the object, and remove the name from the *Name* box in the *Object Properties* dialog. If the *HasResources* flag of the unnamed object was set to *YES*, its resources will disappear from the hierarchy and will not be accessible through the resources mechanism.

Using Tags

Data Tags

To simplify data access for process control applications, a data tag containing a *TagName* and *TagSource* may be assigned to any dynamic parameter or object attribute. Once a data tag is added, the data can be supplied to the attribute by using its *TagSource*. Unlike resources, which are hierarchical, the tags are global and have a flat hierarchy, with all tags visible at the top level. The tags can be accessed by their *TagSource* attribute, without a need to know the hierarchy path as it is the case with resources. Both tags and resources have their own advantages for different types of applications.

Resources are great for handling applications with a large number of instances of similar objects, where it is convenient to show just the names of instances on the top level and display more details when a particular instance is selected. Tags are ideal for process control applications, were the tag's *TagSource* attribute provide mapping between resources of the drawing and fields of a process database. *TagSource* defines the name of a database field which serves as a datasource for the tagged attribute. When a database field changes, an application can update a corresponding tag in the drawing by passing the tag source and new tag value to one of the *SetTag* functions. If multiple attributes share the same tag source, updating the tag will update all such attributes.

The Builder includes a *Tag Browser* which shows all tags of the selected object, or the tags of the whole drawing if no object is selected. If no object is selected and the editing focus is in a child viewport, the tags of that viewport will be shown in the Tag Browser.

A data tag may be added to any attribute or resource object by clicking on the *Add Tag* button in the *Attribute* dialog. The tag's *TagName* attribute may be given any local name that will help the user to identify the tag when browsing. The value of the *TagSource* attribute is used at run-time for accessing the value of the resource object the tag is attached to. Applications that receive data from a database usually use *TagSource* to define the database field to be used as a datasource. A tag also has a *TagComment* attribute that may store any auxiliary information related to the tag.

When a tag is added, its *TagName* and *TagSource* are initially set to the string *"undefined"* and may be edited in the *Data Tag* dialog. The values of the tag's *TagName* and *TagSource* attributes are displayed in the *Tag* field of the *Attribute* dialog as two fields separated by the *'/'* character. To edit the tag, click on the *Edit Tag* button of the *Attribute* dialog. All added tags are shown in a list of tags in the tag browser.

The *Tag Browser* dialog shows the list of tags. Each tag's entry shows its *TagName* and *TagSource* attributes separated by the *'/'* character. Clicking on tag entries selects them, showing the *Data Tag* dialog for editing the tag's attributes and the *Attribute* dialog for editing the attribute object the tag is attached to.

The tags displayed in the tag browser may be sorted by either their tag names or tag sources by using the tag browser's *Sort by* button. The *Filter* filed may be used to display only a subset of tags matching a regular expression that may contain the *?* (any character) and * (any sequence of characters) wild cards. The regular expression will be applied to either the tag names or tag sources as controlled by the *Source/Names* toggle on the right side of the *Filter* field. The toggle also controls the *Selection* field, allowing the user to select a tag by typing its *TagName* or *TagSource*.

The *Display Both/One* toggle switches the view to display both the *TagName* and *TagSource*, or just one of them based on the current *Sort By* setting. If tags are sorted by the tag name, the *TagName* is shown in the *Display One* mode. If tags are sorted by the tag source, the *TagSource* is displayed.

The *Unique Tag Sources/Names* toggle controls if multiple tags with identical tag sources (when sorting by tag sources) or tag names (when sorting by tag names) are shown in the list. By default, the toggle is unchecked and all instances of tags with the same tag source or tag name will be displayed in the Tag Browser. If the toggle is checked, only the first instance of tags with the same tag source or tag name will be displayed.

Editing the value of a single tag in the Builder will not affect the other entries with the same *TagSource*. However, when a tag value is supplied at runtime via its *TagSource*, all value of all tags with the same *TagSource* will be updated.

To see all object's tags, select the object and use either *Object*, *Tags* or the *Tags* toolbar button to bring the *Tag Browser* dialog, see page 342. To see tags of the whole drawing, unselect the object by pressing the *Esc* key. Selecting another object while the Tag Browser is up will show its tags in the Tag Browser.

Refer to the *Tag-Based Data Access and Database Connectivity* chapter on page 60 for details of different ways of using tags for accessing data.

Adding and Deleting Data Tags

Adding a Tag

To add a tag to an object's attribute:

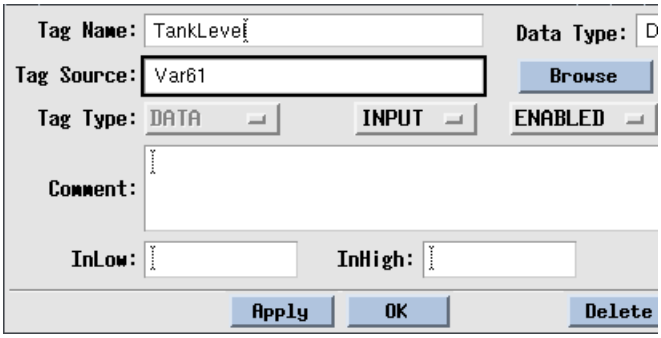
1. Select an object.
2. Use *Object*, *Properties* or use the *Properties* button  to see the attributes of the object.
3. Select an object's attributes by clicking on the attribute's ellipsis button  .
4. Click on the *Add Tag* button.
5. Enter values for the tag's attributes in the *Data Tag* dialog.

To add a tag to a resource:

1. Use *Object*, *Resources* or click on the *Resources* button  to see the resource hierarchy.
2. Locate the attribute resource you want to tag, and click on it to select it.
3. Click on the *Add Tag* button.
4. Enter values for the tag's attributes in the *Data Tag* dialog.

Editing a Tag

To edit a tag attached to an attribute, bring the *Attribute* dialog for the attribute and click on the *Edit Tag* button to activate the *Data Tag* dialog. Alternatively, click on the "*T*" button on the right side of the attribute row in the *Properties* dialog.



The *Data Tag* dialog has entries for editing the *TagName*, *TagSource* and *TagComment* attributes of the tag object. The *Type* field displays the type of the attribute (*D*, *S* or *G*) the tag is attached to.

The *InLow* and *InHigh* entries become active if the attribute the tag is attached to has a range transformation. In this case, the *InLow* and *InHigh* entries of the *Data Tag* dialog provide direct access to editing the *InLow* and *InHigh* parameters of the range transformation. This is convenient when editing tags via the Tag Browser, since the user can assign a datasource and define its value range in one dialog.

Deleting a Tag

To delete a tag attached to an attribute object, follow the following steps:

1. Use *Object*, *Tags* or click on the *Tags* button to see a list of tags.
2. Locate the tag you want to delete and click on it to select it.
3. Click on the *Delete Tag* button in the *Data Tag* dialog.

Alternatively, display the *Attribute* dialog for the attribute or resource the tag is attached to, click on the *Edit Tag* button to display the *Data Tag* dialog and click on the *Delete* button. The "*T*" button on the right side of the attribute row in the *Properties* dialog may also be used to access the attribute's *Data Tag* dialog.

Using Alarms

Alarms may be used to monitor an object's attribute value. A change alarm generates an alarm message when an attribute value changes, and a range alarm generates a message when the value goes outside of the specified range. At runtime, the application code receives alarm messages and processes them according to the application logic. An alarm label may be defined in the Builder to help identify the alarm source at runtime.

The Builder provides an *Alarm Browser* which shows alarms attached to the selected object; if no object is selected, the *Alarm Browser* shows all alarms defined in of the drawing. If no object is selected and the editing focus is in a child viewport, the alarms of that viewport will be shown in the Alarm Browser.

An alarm may be added to any attribute or resource object by clicking on the *Add Alarm* button in the *Attribute* dialog and selecting an alarm type.

The alarms's *AlarmLabel* attribute may be set to any custom string that will help the user identify the alarm when browsing alarms in the Builder, as well as at runtime. The alarm's *Enabled* attribute may be used to enable or disable the alarm.

When an alarm is added, its *AlarmLabel* is initially set to the string *"undefined"* and may be edited in the *Edit Alarm* dialog. To edit an existing alarm, click on the *Edit Alarm* button of the *Attribute* dialog. All added alarms are shown in the alarm browser.

The *Alarm Browser* dialog shows a list of alarms. Each alarm's entry shows its *AlarmLabel*. Clicking on an alarm entry selects it and shows the *Edit Alarm* dialog for editing the alarms's attributes, as well as the *Attribute* dialog for editing the attribute object the alarm is attached to.

The *Filter* field may be used to display only a subset of alarms matching a regular expression that may contain the *?* (any character) and * (any sequence of characters) wild cards.

To see all alarms attached to an object, select the object and use either *Object*, *Alarms* or the *Alarms* toolbar button to bring the *Alarm Browser* dialog, see page 343. To see alarms of the whole drawing, unselect the object by pressing the *Esc* key. Selecting another object while the Alarm Browser is up will show its alarms in the Alarm Browser.

Refer to the *Alarm Object* chapter on page 174 for description of all available alarm types.

Adding and Deleting Alarms

Adding an Alarm

To add an alarm to an object's attribute:

1. Select an object.
2. Use *Object*, *Properties* or use the *Properties* button  to see the attributes of the object.
3. Select an object's attribute by clicking on the attribute's ellipsis button  .
4. Click on the *Add Alarm* button.
5. Enter values for the alarm's attributes in the *Edit Alarm* dialog.

To add an alarm to a resource:

1. Use *Object*, *Resources* or click on the *Resources* button  to see the resource hierarchy.
2. Locate the attribute resource you want to add the alarm to, and click on it to select it.
3. Click on the *Add Alarm* button.
4. Enter values for the alarms's attributes in the *Edit Alarm* dialog.

Editing an Alarm

To edit an alarm attached to an attribute, bring the *Attribute* dialog and click on the *Edit Alarm* button to activate the *Edit Alarm* dialog. Alternatively, click on the "*A*" button on the right side of the attribute row in the *Properties* dialog.

The *Edit Alarm* dialog lists the alarm's attributes. The *AlarmLabel* field may be used to assign a custom alarm name, and the *Enabled* field may be used to enable or disable the alarm. For range alarms, the high and low ranges are also displayed.

Deleting an alarm

To delete an alarm attached to an attribute object, follow the following steps:

1. Use *Object*, *Alarms* to see a list of alarms.
2. Locate the alarm you want to delete and click on it to select it.
3. Click on the *Delete Alarm* button in the *Edit Alarm* dialog.

Alternatively, display the *Attribute* dialog for the attribute or resource the alarm is attached to, click on the *Edit Alarm* button to display the *Edit Alarm* dialog and click on the *Delete* button. The "*A*" button on the right side of the attribute row in the *Properties* dialog may also be used to access the attribute's alarm dialog.

Animating a Drawing

Animation brings the drawing to life by linking a source of data outside the Builder with a resource or tag in the drawing.

When you change the value of an object's attribute, it alters the object's appearance. Animating an object supplies a continually changing series of values to a particular attribute of the object, so the object's appearance is continuously altered. The attribute is addressed via the resource hierarchy; the attribute to animate must appear in the resource hierarchy. The attribute may also be addressed via its tag name.

For example, to animate the radius of a circle named *Circle*, you would use an external data source to provide a series of values to the *$Widget/Circle/Radius* resource object. Executing the animation results in a circle that changes its size. If the radius attribute has been assigned a *radius* tag name, the same animation may be performed by providing values to the *radius* tag.

You may also use a dynamic transformation to animate an object. The simplest way to animate a transformation is by naming its controlling *Factor* by specifying a *Variable Name* in the *Add Dynamics* dialog (the name gets assigned to the *Factor* parameter of the transformation in the *Edit Object Dynamics* dialog). At runtime, you can animate the *Factor* parameter (usually in the range of 0 to 1) to dynamically transform the object.

At run time, the animation is performed by updating resources of the drawing with new values using the GLG Standard API. The C/C++, Java, C#/.NET or ActiveX version of the API is used depending on the choice of the GLG container used to load the drawing. **At design time**, the GLG Builder provides a Run mode and a data generating tool (called *datagen)* for prototyping the drawing right in the Builder. A custom proto DLL can also be used to supply real-time data for prototyping, see the *Custom Run Module DLL* section on page 290.

Use *Run*, *Start* to prototype the drawing with test data and enter the run command for *datagen*. When prototyping a widget drawing loaded from the palette, the run command is usually preset to the correct value. To animate a custom drawing or resource, change the run command. To animate tags, use the *-tag* option. For the full syntax for *datagen*, see the *GLG Programming Tools and Utilities* chapter. For an example of using *datagen* to animate a drawing, see the *Drawing a Simple Example* section of this chapter. Refer to the *The Data Generation Utility* chapter of the GLG Programming Reference Manual for more details.

Reusing Objects, Attributes, and Transformations

The Builder provides a variety of ways to replicate the elements of a drawing. You can reuse entire objects, attribute values, or transformations.

Reusing an Object

The Builder provides a variety of methods for making copies of the selected object. These methods include:

- Saving objects to a drawing file, and loading an object from a drawing file into the current drawing.
- Cutting and pasting the object via the clipboard.
- Cloning the object, which copies it and applies an offset or a transformation.
- Saving objects to a drawing file and using them as *sub-drawings* with *reference objects* (refer to the *Reference* section of the *Advanced Graphical Objects* chapter for more details).
- Using an objects as a template of a reference object for replicating instances of it in the drawing (refer to the *Reference* section of the *Advanced Graphical Objects* chapter for more details).
- Saving objects to a drawing file and adding them to the Custom Object palette.

Copying or cloning an object that is a named resource adds a duplicate branch to the resource hierarchy. If the duplication causes a naming conflict, rename the copy.

To copy more than one object in a single operation, first create a group object that contains the objects, using *Arrange*, *Group*. When you have finished the operation, select the group and use *Arrange*, *Explode, Object* to delete the group object and restore the independence of the objects.

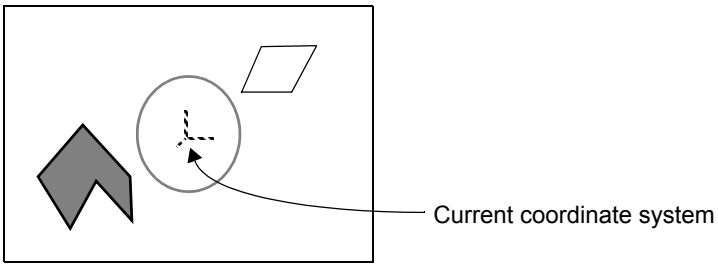
A temporary group may also be used by either selecting objects by dragging a rectangle with the mouse, or by using *Ctrl-click* to add or delete objects from the selection. The temporary group is automatically destroyed when it is unselected.

If you are copying the objects because you want to create a set of identical objects that can be edited in one place as a single object, you should consider using a series, reference or other advanced object; see page 265.

Saving an Object to a File

You can save the selected object to a file, as a GLG drawing. Loading the saved object adds it to the current drawing, adding all its attributes to the object hierarchy of the current drawing. With this method, the object's availability persists across Builder sessions, because the object is in an external file. Use *File*, *Save Object* to create a new file that contains the selected object, and use *File*, *Load Object* to add the object to the current drawing; see page 298.

When you load an object from a file, the object is placed by using its coordinates in the coordinate system of the current level of the hierarchy. For example, if you select a viewport and then load a circle with a center at 0,0,0, the circle is drawn in the middle of the viewport.



Cutting, Copying, and Pasting an Object

You can cut or copy an object to a temporary storage area, and then paste it into the drawing. The clipboard holds a single object, which stays on the clipboard until you cut or copy another object. With this replicating method, the object can be copied from one level of the traversal hierarchy to another.

Use *Edit*, *Cut* and *Edit*, *Copy* to copy the selected object to the clipboard; cutting it deletes the original object while placing it on the clipboard, but copying it leaves the original intact and places a copy on the clipboard.

Use *Edit*, *Paste* to retrieve the object. When you copy and then paste an object, the object is placed directly over the selected object, at the current level of the hierarchy. You can then drag it into position. The *Paste Clone Type* option of the *Options* menu controls a constrain type of the copy's attributes.

The *Cut, Copy* and *Paste* toolbar icons may be used for fast access.

Cloning an Object

You can clone an object, which adds another copy of the object to the drawing. Cloning gives you more control over the nature of the copy; you can constrain the clone, place it using a specified offset from the original, or transform it as it is created.

The Builder provides several varieties of cloning:

- A *full* clone is just a copy of the object, with no constraints. This is the default for copy and paste operations.
- A *constrained* clone is a copy of the object, with all attributes and control points of the copy constrained to the corresponding attributes or control points of the original object.
- A *strong* or *weak* clone is a constrained copy of the object. The *Global* attribute determines what attributes of the clone are constrained.

Each attribute or control point has a *Global* attribute, which appears in the *Attribute* dialog. This attribute has three possible values that interact with the strong and weak clone types to provide a range of possibilities for constraining while cloning:

- *Global* constrains the clones to the original for both a strong and a weak clone.

- *SemiGlb* constrains the clones to the original in a strong clone, but not in a weak clone.
- *Local* does not constrain the clones to the original for either a strong or weak clone. This is the default.

To control the position where the clone appears, use *Edit*, *Clone Offset*. To define a static transformation to apply to the clone as it is created, using *Edit*, *Clone Transformation*. The clone transformation may use rotational or other offsets.

Adding an Object to the Custom Object palette

To add an object to the *Custom Object* palette, simply save it in the *widgets/custom_objects* directory. The object will automatically be added to the *Custom Objects* palette when the builder is restarted or the palette is re-opened.

It's also possible to add your own palettes of objects to the Builder. Refer to the *Palettes* section of the *GLG Graphics Builder Menus* chapter for more details and options.

Marking Transformations, Rendering Attributes and Other Objects

The Builder includes a marking mechanism for marking attributes and resources, as well as transformation, aliases, custom properties, rendering and text box attributes, color and font tables, and other objects. Marking attributes is used for constraining to a marked attribute, as well as for reusing its value. Marking transformations, aliases, custom properties, rendering attributes and other objects is used for replicating these objects as well as for attaching constrained transformations or rendering attributes to several objects. The marking mechanism parallels the copy and paste mechanism, but its clipboard is entirely independent. The marking of attributes, transformations, aliases, custom properties and other objects are completely independent, but have similar uses.

Marking an attribute, transformation or other object for copying is similar to cutting and pasting with a clipboard; the marked entity stays marked until you mark another.

Marking an Attribute

The Builder lets you make the value of an attribute available for constraint or reuse with another attribute, if the two attributes use the same data type. You can mark up to five separate attribute values for reuse.

To mark an attribute value for reuse:

1. Select the object with the value you want to mark.

2. Use *Object*, *Properties* or click on the *Properties* button  in the toolbar to show the *Selected Object Properties* dialog. This dialog shows a list of attributes, with a for attributes that can be modified.

3. Click on the to see the *Attribute* dialog.

4. Select the *Mark* button. The Builder prompts you with a list of marks, ranging from *Mark0* to *Mark4*. Make a note of the marks you assign; they are not named.

5. Click on the mark to which you want to assign the attribute's current value.

The attribute value has been marked, and assigned to the mark number you specified.

Using Marked Attributes

There are two ways to use a marked attribute value: you can constrain the attributes, or set the second attribute's value to the marked value. To use a marked attribute:

1. Show the list of attributes for the object:

   a. For an attribute, use *Object*, *Properties* to show the *Selected Object Properties* dialog, and click on the ... to see the *Attribute* dialog.

   b. For a control point, use *Shift+click* to see the *Control Point* dialog.

2. Click on either the *Constrain One*, *Constrain All* or the *Use Object* button. The Builder prompts you with *Use Resource* and *Use Marked* buttons. If *Constrain All* is used, not only the attribute itself, but also all attributes constrained to it will be constrained to a new object. (*Constrain One* is enabled only if activated in the *glg_config* file.)

3. Click on *Use Marked*. The Builder prompts you with a list of marks, ranging from *Mark0* to *Mark4*.

4. Click on the mark to use for the new attribute value.

The attribute value from the mark is applied to the current attribute.

When the drawing is reloaded, the marked attributes point to the attributes of the old drawing. Therefore, you can still reuse their values, but can't use them for constraining.

Unconstraining an Attribute

To unconstrain an attribute:

1. Show the list of attributes for the object:

   a. For an attribute, use *Object*, *Properties* to show the *Selected Object Properties* dialog, and click on the ... to see the *Attribute* dialog.

   b. For a control point, use Shift+click to see the *Control Point* dialog.

2. Click on either the *Unconstrain* button. If an attribute has a transformation attached to it, the *Attribute Clone Type* entry in *Options* controls cloning type of the transformation's attributes. The default is *Constrained Clone*, which means that while the attribute itself is unconstrained, the transformation's parameters will still be constrained.

Marking a Transformation

You can mark a transformation in a way that parallels the marking of an attribute; however, the two mechanisms are entirely separate.

The Builder lets you copy the definition of a transformation from one graphical object to another. You can also copy the definition from one data object to another, if the two data objects use the same data type. Marking a transformation for copying is similar to cutting and pasting with a clipboard.

In the object hierarchy, reusing a transformation definition produces another independent transformation object, attached to the object you select. The parameters of two transformation objects may be constrained by the reuse operation as described below, producing "linked" transformations changing synchronously and transforming several objects by changing just one controlling parameter.

To mark a transformation definition:

1. Select the object to which the transformation is currently attached.

2. Show the list of transformations for the object:

   a. For a graphical object, use *Object*, *Edit Dynamics* or click on the *Xform* button in the *Status Panel* to see a list of transformations.

   b. For an attribute, use *Object*, *Properties* to show the *Selected Object Properties* dialog, and click on the ... to see the *Attribute* dialog (properties that have a transformation attached are annotated with an *X* on the right side of the property in the Properties Dialog). Finally, click on the *Edit Dynamics* button.

   c. For a control point, use *Shift+click* to see the *Control Point* dialog, and click on the *Edit Dynamics* button.

3. In the Transformation List dialog, use the *Mark Object* button to mark a single transformation, or use *Mark List* to mark all transformations in the list.

The transformation(s) you marked can be reused for any graphical or data object with the same data type.

To reuse marked transformations:

1. Select the object to which you want to attach the marked transformation(s):

   a. For a graphical object, use one of the transformation options on the *Object* Menu or the Toolbar (*Transform Points*, *Add Static Transformation*, or *Add Dynamics*).

   b. For an attribute, use *Object*, *Properties* to show the *Selected Object Properties* dialog, and click on the ... to see the *Attribute* dialog (properties that have a transformation attached are annotated with an *X* on the right side of the *Properties* dialog). Finally, click on the *Add Dynamics* button.

   c. For a control point, use *Shift+click* to see the *Control Point* dialog, and click on the *Add Dynamics* button.

2. For an object or control point, click on the *Transformation Type* list, select *Use Marked* and the cloning type. Selecting *Full Clone* creates a new transformation with the same attribute values but without constraining. Selecting the *Constrained* cloning type creates a new transformation with attribute constrained to the original transformation. The *Strong* clone type constrains attributes of the transformation whose *Global* flag is *Global* or *Semi-Global*, and *Weak* clone constrains only *Global* attributes.

   For attributes, click on the *Use Marked* button and then select a clone type from a popup menu.

3. For an object or control point, click on the *Apply* button to attach the marked transformation(s).

The marked transformation(s) stay marked until you mark another transformation. When the drawing is reloaded, the attributes of the marked transformation point to the attributes of the old drawing. Therefore, you can still reuse the transformations using *Full Clone*, but can't reuse constraints implied by the rest of the cloning types.

Marking Rendering and Text Box Attributes, Fonttables and Light Objects

Rendering and *Text Box Attributes*, as well as viewport's *Fonttables* and *Light Objects* can be marked for reuse by using the *Mark* button in the *Properties* dialog for these objects. To reuse the marked object, select *Edit, Add or Use Marked Object*, then select an option that matches the type of the marked object.

The *Options, Attribute Clone Type* menu controls constraints of the new instance's attributes: if it is set to *Constrained Clone*, all attributes will be constrained to the corresponding attributes of the original object. For example, to add constrained copies of rendering attributes to several objects, add *Rendering Attributes* to one object, mark the *Rendering Attributes* object, create a temporary group containing the objects, select *Options, Attribute Clone Type, Constrained Clone*, then select

Edit, Add or Use Marked Object, Rendering Attributes. It will attach constrained copies of rendering attributes to all objects in the group: changing a rendering attribute of one object will affect rendering of all these objects.

Marking Aliases

Marking aliases is a convenient way of copying a list of aliases from one object to another.

For marking aliases attached to an object, select the object, then select *Aliases*, *Mark Aliases* from the *Object* pull-down menu. To add marked aliases to another object , select a new object, then choose *Aliases*, *Add Marked Aliases* from the same pull-down menu.

Marking Custom Properties

Marking custom properties provides a way to copy a list of custom properties from one object to another.

For marking custom properties attached to an object, select the object, then select *Custom Properties*, *Mark Custom Properties* from the *Object* pull-down menu. To add marked custom properties, select another object, then choose *Custom Properties*, *Add Marked Properties* from the same pull-down menu.

The custom property lists can contain lists of lists of custom properties. For example, an option menu object may have a list of custom properties that contains another list named *InitItemList*. The *InitItemList* contains items to be displayed in the option menu. To mark this inner list, select the option menu object, select *Custom Properties*, *Edit Custom Properties* from the *Object* pull-down menu, select *InitItemList* with a single mouse click and press the *Mark List* button. To add the marked list of custom properties as a list to another object, select a new object, then select *Custom Properties, Add Custom Properties, Add Marked List* from the *Object* pull-down menu.

Controlling the View

A GLG drawing is a set of three dimensional objects defined in limitless space. However, the Builder renders these objects in two dimensions on the screen.

To present a drawing, the Builder applies a set of transformations to the objects in the drawing. To get a different view of the drawing, you can change some of the parameters for these transformations, including the view projection, scale, coordinate system, and center.

For example, the Builder's main view shows the X and Y axes, with the Z axis perpendicular to the screen. In this projection, the control points of objects you create in the main view are positioned only with X and Y coordinates, with their Z coordinates set to zero. Using the mouse to edit objects in the default view only affects the X and Y values for the objects; the Z dimension is unaffected.

The Builder saves the current view for a viewport as part of the drawing. The view has no effect on the definition of the objects in the drawing, however.

To change your view of a drawing, use the options on the *View* Menu, or the *Pan*, *Zoom*, and *Rotate* controls.

Changing the View Projection

The Builder uses a view projection to transform the drawing from its three dimensional definition into the two dimensional rendering that appears in the drawing area. Your access to objects is limited by the view projection; the objects you draw are drawn in that plane. To get access to other planes for drawing, you change the view projection.

The Builder includes a set of six predefined view projections, but you can also define and use your own. Use the options on the *View*, *Set View* submenu to switch between predefined projections; see page 310.

To define your own projection, use either *View*, *Adjust View* (see page 311) or the buttons and sliders on the lower left side of the Builder window. The menu option gives you precise control over the projection, but the sliders let you change the projection interactively.

Note that these view sliders are stateless. The middle of the slider range always represents the drawing view *before* the slider is activated. This means that wherever you move the slider, it returns to the middle of its range when you release the mouse button, ready for the next move.

Customizing the View Projection

If you want to switch between several projections, you may want to save the transformation for the projections into files. When you want to view a drawing using a particular projection, you load the view projection. Use *View*, *Save Viewing Transformation* and *View*, *Load Viewing Transformation* to create and load a projection file; see page 312. Because the Builder saves the current view projection along with the drawing, you only need this option if you switch projections frequently.

To save or load a viewing transformation of a particular viewport, move editing focus into it using the Set Focus button or go down into it using the Hierarchy Down button.

To transform a viewport dynamically from a program, add a parametric transformation to the viewport as a view transformation. Changing parameters of the transformation will change the view projection. Alternatively, the program may use the *GlgSetZoom* method.

Viewing Using Different Coordinate Systems

By default, the Builder uses the coordinate system of the entire drawing for specifying points and transformations. However, you can view the drawing using the coordinate system of any object in the drawing to clarify the relationships between the objects. For example, changing the coordinate system allows the rotation of objects in different coordinate systems.

Use the options on the *View*, *Coordinate System* submenu to select a coordinate system. The effect of these options depends on the content of the drawing and the object you select. For example, consider a drawing that contains a group of three polygons; such a drawing has separate coordinate systems for each polygon, for the group object, and for the viewport. For this drawing, all the *View*, *Coordinate System* submenu options are applicable. However, for a drawing containing one polygon, only the *Object* option applies.

Changing the Viewing Area

To control the scale of the drawing, use the options on the *View*, *Zooming* submenu, or the *Zoom* and *ZoomTo* buttons in the *Control Panel*. 

To use a different point as the center of the Builder window, use the *View*, *Pan To* option. Alternatively, you can use scrollbars or *Ctrl-click* in the drawing to drag it with the mouse.

Using Advanced Objects

The Builder includes certain objects that can be used for specialized functions. They simplify the construction of complex drawings.

The functions and the objects that provide them are:

Function	**Object type**
Associate objects together, either temporarily or permanently	*Group*
Create a set of replicated objects from a template, spacing them equally along a path	*Series*
Create a set of replicated objects from a template, arranging them on a grid	*Square Series*
Provide a container with one control point for holding objects	*Container, Reference*
Replicate instances of a template object	*Object Reference*
Replicate instances of a sub-drawing	*File Reference*
Create a line with a defined number of points, or a segmented line	*Polyline*
Create a patched surface	*Polysurface*
Create a frame to which other objects may be constrained	*Frame*
Connecting points or nodes with a recta-linear or arc path	*Connector*
Rendering gradient fill, arrowheads, cast shadows and fill dynamics	*Rendering*
Rendering a box around a text object	*Box Attributes*
Scrolling matching attributes of a collection of objects	*History*
Rendering GIS map data	*GIS Object*
Attaching custom properties	*No dedicated object type*
Defining logical resource names	*Alias*

Associating Objects Together

To associate objects, you create a group object that acts as a container. A *Group* object does not contribute to the drawing visually; it is primarily an editing tool with three main purposes:

- To keep a set of otherwise unrelated objects together. The members of a group are spatially associated, and move together.
- To let you act on a set of objects collectively. The action you perform is applied to every member of the group.
- To apply animation to a set of objects.

Use *Arrange*, *Group* to create a group. The Builder prompts you to draw a rectangle that touches or encloses all the objects to be included in the group. A group object does not appear as a visible graphical object, but the control points of objects in a group appear as hollow squares. Clicking on any member of a group selects the group.

To release one object from a group, use the options on the *Arrange*, *Explode* submenu. To release all the objects from a group and delete the group object, select it and use *Arrange*, *Explode*, *Object*.

Editing Group Members

Although the members of a group are associated, they can still be edited individually. You can move the control points of a group member by dragging or editing them.

For more intensive editing, use the options on the *Traverse* and *Arrange* Menus to get access to an individual object:

- Open the group using *Traverse*, *Hierarchy Down*. By navigating to the appropriate level of the hierarchy, you can edit the individual members of the group. Use *Traverse*, *Up* when you have finished.
- Release or add members to the group using *Arrange*, *Remove Object from Group* and *Arrange*, *Add Object to Group*; see page 318.
- Destroy the group and replace it with its members by using the *Arrange*, *Explode* submenu; see page 320.
- Edit attributes that are common to the group members using *Traverse*, *Edit All* or the button on the group's *Properties* dialog. The changes you make are applied to every member of the group; see page 316.
- Enter group editing mode using *Traverse*, *Select Next* or the button on the group's *Properties* dialog. You can edit the group member you select, without having to traverse the hierarchy; see page 315.
- For nested groups, move directly to the lowest level of grouping using *Traverse*, *Select Bottom* or the button on the group's *Properties* dialog; see page 315.

Temporary Groups

A temporary group may be used for quick editing of several objects. A temporary group may be created by dragging a rectangle around objects in the drawing with the mouse; all objects that intersect a rectangle will be included into the group. Alternatively, *Ctrl-click* may be used to add or

delete objects from the selection. The *Edit* and *Arrange* menus contain additional options for creating temporary groups. The *Permanent Group* toggle in the *Arrange* menu may be used to change the type of the group from temporary to permanent and vice versa.

The temporary group is automatically destroyed when it is unselected.

Generating Objects from a Template

To create a set of objects that share characteristics, you can generate objects from a template instead of cloning or copying an original. With generated objects, you can change the whole set just by editing the template, and position the set automatically.

The *Series*, *Square Series*, and *Reference* objects all consist of a set of instances and a template that controls the characteristics of the instances. The instances are created dynamically from the template. The number and positioning of the instances depends on the object type.

To create a series, square series, or reference object, you first create and select an object to act as the template. When you create the instances, the Builder moves the template and the instances into a new object. You can get access to the template using the *Traverse*, *Hierarchy Down* menu option.

The attributes of the template override any changes you make to an instance. To customize the instances, explode the series using *Arrange*, *Explode*, *Object*; see page 320. Then edit the attributes of the members of the resulting group.

The Series Object

A series presents a set of instances arranged along a line.

The number of instances is controlled by a factor, and their positions are determined by distributing them along a line or path. The instances are named using the template object name and an index; for example, a template named *Rect* with a factor of 3 creates three instances named *Rect0*, *Rect1*, and *Rect2*.

A *Series* object has four control points. Two points control the location of the line along which the series instances are distributed. A third point controls the position of the first instance's origin; it is constrained to one of the path points. A fourth point is visible only if you use *Traverse*, *Hierarchy Down* to see the series template; this point provides an alternative way to position of the first instance's origin.

To create a series:

1. Select an object to use as a template.

2. Select *Object*, *Properties* to name the object, and to set the attributes of the template object so that its instances will inherit appropriate characteristics.

3. Select *Object*, *Create*, *Series* to create the instance objects.

4. Click on two points to specify a path on which to arrange the series instances.

5. Enter a factor to specify the number of instances to create.

To edit the template object, use *Traverse*, *Hierarchy Down*. When you finish editing, use *Traverse*, *Up* to see the instance objects.

The Square Series Object

A square series is a series with its instances organized into rows and columns. The number of rows and columns determines the number of instances in the square series.

A *Square Series* object consists of a template and a set of generated instances. The instances are named using the template object name and an index; for example, a template named *Rect* with two rows and two columns creates four instances named *Rect0*, *Rect1*, *Rect2*, and *Rect3*.

To create a square series:

1. Select an object to use as a template.

2. Select *Object*, *Properties* to name the object, and to set its attributes so that its instances will inherit appropriate characteristics.

3. Select *Object*, *Create*, *Square Series* to create the instance objects.

4. Click on the origin for the square series, and click on two points to specify two vectors from the first point; they control the arrangement of the series instances.

5. When prompted, enter the number of rows, then the number of columns. These values specify the number of instances to create.

To edit the template object, use *Traverse*, *Hierarchy Down*. When you finish editing, use *Traverse*, *Up* to see the instance objects.

The Reference Object: Containers and SubDrawings

A reference is essentially a series with a single element. Like the series, it uses a template object, but only one instance (the *Reference* object) is created.

A reference is most useful when there is a need to replicate an object throughout a drawing or multiple drawings. It may also be used to implement subdrawing or object dynamics, changing the displayed object based on some condition. There are several types of reference objects:

Container

A *Container* object is used to encapsulate a set of objects, protect them from accidental editing and provide a control point which may be used to move or constrain objects. When a container is copied, its template object is copied as well. To create a container, select an object to use as a template, click on the *Container* icon in the *Object Palette* and click in the drawing to position the container.

Included SubDrawing

An *Included SubDrawing* is used to replicate instances of a template in a drawing. Copying or cloning the subdrawing creates a new instance that uses the same template. Editing the template modifies all the instances of the subdrawing in the drawing.

To create instances of the template, first create an *included subdrawing:* select an object to use as a template, click on the *SubDrawing From Object* icon in the *Object Palette* and click in the drawing to position the subdrawing. If the template contains several named objects used as icons for object dynamics, enter two colon-separated resource paths, to one of the objects (*ObjectPath*) and its anchor point (*OriginPath*), and press *OK*. To display the whole template press *OK* without entering *ObjectPath*.

File SubDrawing

A *File SubDrawing* is used to replicate instances of another drawing used as the subdrawing's template. Editing the template drawing changes all instances of it in other drawings. To create a file subdrawing, click on the *SubDrawing From File* icon in the *Object Palette*, click in the drawing to position the subdrawing and define the drawing file to be used as the template. If the template drawing contains several named objects used as icons for object dynamics, enter colon-separated resource paths to one of the objects (*ObjectPath*) and its anchor point (*OriginPath*), and press *OK*. To display the whole template drawing press *OK* without entering *ObjectPath*.

Palette SubDrawing

The *Palette SubDrawing* uses some object in the drawing (palette) as a template. To create a palette subdrawing, select *Object, Create, SubDrawing, SubDrawing From Palette* and click in the drawing to define the subdrawing's position. If the palette contains several named objects used as icons for object dynamics, enter two colon-separated resource paths, to one of the objects (*ObjectPath*) and its anchor point (*OriginPath*), and press *OK*. To display the whole palette, press *OK* without entering *ObjectPath*. Edit the subdrawing's properties and enter palette object's resource path in the *SourcePath* attribute.

To edit the template of a container or subdrawing object, select it and use *Traverse*, *Hierarchy Down*. For *File SubDrawings*, traversing down loads the referenced drawing, and traversing back up saves it. For *Palette SubDrawings*, traversing down performs *Hierarchy Down* into the palette object. To delete the subdrawing's wrapper and replace it with the template, use *Arrange*, *Explode*, *Object*; see page 320.

To adjust the position of the subdrawing's (or container's) graphics relative to its control point, you can move their template to change its position relative to the center of the drawing or the *Origin* point (a round marker). You can also adjust the anchoring by pressing *Shift+Control* and moving the subdrawing's control point relative to its graphics, without traversing down to edit the template.

Containers and subdrawings may either be scalable or have fixed size, as controlled by their *FixedSize* attribute.

Containers and subdrawings may be used as nodes connected with *connector* objects constrained to their control point.

Creating Animated Lines and Surfaces

For animated lines and surfaces, use the **polyline** and **polysurface** objects. A polyline is a line or set of line segments, with a defined number of points. A polysurface is a set of polygons. These are special objects used in graphs that may be animated by using a history object to address their points or segments; see page 272.

The polyline and polysurface objects use two templates. The *Marker* template is used for the control points of the objects. The *Polygon* template controls the line segments of the polyline, and the surface polygons of the polysurface. Each instance of the object is named using the template object name and an index. For example, for a two-segment polyline, there are three marker instances (*Marker0, Marker1*, and *Marker2*) and two line instances (*Polygon0 and Polygon1*).

The Polyline Object

A polyline is a specialized series that consists of a line or set of line segments and points.

To create a polyline, click on two points to specify the beginning and end of the polyline. The Builder prompts you for the factor, which controls the number of points along the line.

The number of segments in the polyline is controlled by its *Segments* attribute.

To edit the *Marker* template object, use *Traverse*, *Hierarchy Down*. When you finish editing, use *Traverse*, *Up* to see the instance objects. To edit the *Polygon* template, use *Object*, *Resources*.<

The Polysurface Object

A polysurface is a specialized, three-dimensional series object. It defines a set of surface patches.

To create a polysurface, click on a point to specify the center of the polysurface, and click on two points to specify two vectors from the center point. The Builder prompts you for the number of rows and columns in the surface; these values control the number of surface polygons.

To edit the *Marker* template object, use *Traverse*, *Hierarchy Down*. When you finish editing, use *Traverse*, *Up* to see the instance objects. To edit the *Polygon* template, use *Object*, *Resources*.

Attaching Objects to a Frame

A frame is a configuration of points that can be adjusted and positioned as a single object. Each segment of the frame is populated with frame points that let you use the frame's geometry to position objects by constraining them to the frame points. The *Reference* object may be used as a container to hold several objects connected to a frame, in which case the *Reference's* control point is attached to the frame.

There are five types of frames, which provide different configurations of frame points:

- A **point frame** allows anchoring to a single point.
- A **line frame** allows anchoring to points along a line.
- A **2D frame** allows anchoring to points inside a parallelogram.

- A **3D frame** allows anchoring to points inside a parallel prism.
- A **free frame** consists of arbitrarily placed points.

A frame has two sets of points: its own control points for controlling its geometry, and the constrained frame points for use by other objects. Use *Options*, *Show Frame Points* to toggle between the control points and the frame points; see page 358. Note that for the free frame and the point frame, the frame points and the frame's own control points coincide.

Connecting Objects with a Path

The *Connector* object can be used to connect other objects with a recta-linear or arc path. To create a connector, select one of the Recta-Linear Edge buttons, or an Arc Edge button, then click in the drawing to define the connector's points. The connector connects its points with either a recta-linear or arc path.

The end points of the connector can be constrained to other objects. For example, you can use reference objects as nodes and constrain the end points of a connector to the control points of references. The connector will maintain the connecting path when the nodes are moved.

The recta-linear connector also provides access to its *constrained points*. These points can't be edited since their position is defined by the connector's control points. However, they may be used to constrain other objects to the middle point of a connecting path. Use *Options*, *Show Frame Points* to toggle the selection display between the control points and the constrained points. Note that some constrained points (at the start and end of each path segment) coincide with control points.

Defining Extended Set of Rendering and Text Box Attributes

The Rendering object is used to define optional rendering attributes, such as gradient fill, cast shadow, arrowheads and fill dynamics. For the text object, the Box Attributes object may also be used to define attributes of the box drawn around the text.

To add rendering attributes to an object:

1. Select the object to which you want to add rendering attributes.

2. Click on the *Add Rendering* button at the end of the object's properties. If the Rendering object has been already added, the button name will be Edit Rendering: in this case, click on it to edit rendering attributes.

3. Change rendering attributes to define gradient fill, cast shadow, arrowheads and fill dynamics parameters.

4. To delete rendering attributes, click on the *Delete Rendering* button in the Rendering Properties dialog.

To add text box attributes to a text object:

1. Select the text object to which you want to add the box to.

2. Click on the *Add Box Attributes* button at the end of the object's properties. If the Box Attributes object has been already added, the button name will be *Edit Box Attributes*: in this case, click on it to edit the box attributes.

3. Change the box attributes to define the text box's appearance.

4. To delete box attributes, click on the *Delete Box Attributes* button in the Box Attributes' Property dialog.

Scrolling Attributes of Objects with Index-based Names

The **history** object provides a way to animate resources that use numeric values in their names. When such a resource is animated, the history object provides access to each value in turn. The most obvious application for the history object is for series objects, polylines, and polysurfaces, since these objects append a numerical index to names of its template's instances. The history object can also address named resources in a group if they use the same type of naming convention and the group object's *HasResources* flag is turned on.

To create a history object:

1. Select the object to which you want to add a history object.

2. Select *Object*, *Add History.* The Builder prompts you for the resource name.

3. Enter the resource name, replacing the numeric part with a percent (%) sign. If the resource is not a direct child of the object you selected, specify a relative "path" to the resource.

4. The Builder adds a resource named *EntryPoint* to the resource hierarchy. The *History* object is not represented visually; its existence is indicated by *EntryPoint*.

For example, consider a series named *S* with its HasResources flag set to YES and with a template named *Triangle*. Its instances are named *Triangle0*, *Triangle1*, *Triangle2*, and *Triangle3*. To animate the fill color of the instances, you add a history object to *S*, and specify *Triangle%/FillColor* as the resource name. The *datagen* command line to animate the fill color of the instances would send data to the *EntryPoint* object defined at the same level as the *Triangle* object (*$Widget/S/EntryPoint*).

To animate the resource, provide input data to the *EntryPoint* resource. Each member of the numeric series is updated in turn.

To access a list of history objects attached to the selected object, use *Object*, *Edit History* menu option or click on the *Hist* button in the *Status Panel.* To delete a history object, select the parent object and use *Object*, *Delete History.* The Builder deletes the first history object in the list. You can also use the *Delete* button at the bottom of the *History List* to delete the highlighted history object from any position.

If you explode a series, polyline, or polysurface object that has a history object, the group that remains after exploding it retains the history object. Exploding that group discards the history object.

Rendering GIS Map Data

The **GIS Object** provides a way to utilize functionality of the GLG Map Server in a GLG drawing, embedding GIS map displays into GLG drawings in both the Builder and an application. The GIS Object transparently handles all aspects of low-level map-server interaction to display, zoom and pan the map.

To create a GIS Object, select a GIS Object button in the Object Palette, then click in the drawing area to define two points that specify the rectangular area to be used for the map display. Then enter the dataset file, which tells the Map Server what GIS data to render. The GIS Object provides attributes to control the projection, center and extent of the map, as well as the layers to be displayed on the map.

To set the GIS Zoom Mode, select the GIS Object and use the *Arrange, GIS Zoom Mode, Set as parent viewport's GIS Object* menu option. In the GIS Zoom Mode, the zoom and pan controls of the Builder zoom and scroll the map displayed in the GIS Object instead of zooming and scrolling the drawing. Refer to the *Integrated Zooming and Panning* chapter on page 45 and the *Integrated GIS Object, GIS Rendering and GIS Editing Mode* chapter on page 46 for more information.

Adding Custom Properties to Objects

Custom Data Properties may be used to associate application-specific information with an object. This information is persistent and may be saved with the drawing, or accessed using the resource access functions. The available data types of custom properties (*D*, *S* and *G*) match the data types of object attributes and may be used to keep information in the form of numerical values or strings.

To add custom data properties to an object:

- Select an object to which you want to attach the custom data properties to.
- Select *Object*, *Add Custom Property* and select a *D*, *S* or *G* property type.
- The Builder adds a custom property, displays a *Custom Properties List* and a dialog for editing the property. Enter the property name and value. The property name will be used for accessing the property.
- Add more properties using the above steps. When finished, use the Resource Browser to browse them as resources.

The *Data* button of the *Status Panel* may be used to access a list of custom properties of the selected object.

Defining Logical Names using Aliases

An alias object may be used to define logical names for arbitrary resource hierarchies. For example, it may define a logical "ValueHighlight" name for accessing the "Group1/Object3/FillColor" resource hierarchy. The application can then access the resource using a logical resource name instead of a complete path name. The alias can also be used to create convenient shortcuts for long path names.

To define an alias for a resource hierarchy:

- Select the object whose resources you want to reference using the alias. The resource path will be defined relative to this object. The *HasResources* flag of the object must be set to YES to enable access to its named resources.
- Select *Object*, *Add Alias*. The Builder adds an alias, displays the *Alias List* and the attributes of the added alias.
- Enter the logical name into the Alias attribute field.
- Enter the resource path you want to assign to the alias into the *Path* attribute. To define the path using the *Resource Browser*, select the ellipsis button ... for the *Path* attribute, select the resource and press *Select*.
- Repeat the above steps to add more aliases. When finished, use the *Resource Browser* to check the aliases.

The *Alias* button of the *Status Panel* may be used to access a list of aliases defined for the selected object.

The *Mark Object* and *Mark List* buttons in the *Alias List* dialog may be used to mark the currently selected alias object or the whole alias list for reuse. To add marked aliases to a different object, select the object and use *Object, Aliases, Add Marked Aliases* from the main menu.

Drawing a Simple Example

The following example shows how to draw and animate a couple of valves. It incorporates several of the most typical tasks encountered in building and animating a GLG drawing.Where the instructions below use the choices available from the Builder menus, you can also use toolbar and object palette buttons.

Attribute Animation

The first task is to create a drawing for the valve handle and to animate some of its simple attributes.

1. Create a viewport and name it "$Widget." Use *Object, Create, Viewport* to create the viewport (or click on the *Viewport* button in the object palette), and use the mouse to specify the viewport's corner points. Use *Object*, *Properties* or click on the *Properties* button on the toolbar to show the *Selected Object Properties* dialog. You can use this dialog to specify the viewport's name.

2. Select the viewport with the mouse (if you have just specified its name, it is already selected). Use *Traverse*, *Hierarchy Down* or the down arrow button in the hierarchy controls to go "down" into the viewport.

3. Inside the viewport, create a polygon that looks like a valve handle to you and name it "handle." Use *Object, Create, Polygon* to create the polygon, and use the mouse to specify the polygon's points. Press the right mouse button when you are finished specifying points. You can use the *Selected Object Properties* dialog to specify the polygon's name.

4. Select *Traverse, Up*. Open the resource browser with *Object, Resources*, or with the *Resources* toolbar button  , and note that both the *$Widget* name and the *handle* name are on the same level of the resource hierarchy.

5. Close the resource browser, select the viewport, bring up the *Selected Object Properties* dialog, and set the viewport's *HasResources* flag to YES.

6. Now check the resources again. Open up the resource browser again, and note that while the *$Widget* resource still appears at the top level of the hierarchy, the *handle* resource is gone. If you double-click on the *$Widget* name, its subsidiary resources appear, now including the missing *handle* resource. This illustrates the use of the *HasResources* flag: it defines where in the resource hierarchy an object's children appear.

7. Double-click on the *handle* resource, and observe the default polygon attributes (*LineWidth*, *FillColor*, and so on) below it. Unlike named resources, these default attributes appear below the polygon object whether or not the polygon's *HasResources* flag is set to YES.

8. Select *Run, Start* and at the run prompt, issue the command:
   ```
   $datagen -sin d 0 10 $Widget/handle/LineWidth
   ```

9. Select *Run, Stop* to stop the line width animation. Select the viewport, then *Traverse*, *Hierarchy Down*. Select the *handle* polygon, and bring up the *Selected Object Properties* dialog.

10. Next to the *LineWidth* attribute, there is a button labeled ... . The button indicates that the attribute name refers to an object. If you press it, an object dialog appears, where you can type a name for the object. Give this the name "handle_width."

11. If you were to examine the resource hierarchy now, you would see that the *handle* and *handle_width* resources are on the same level as each other. To make the *handle_width* appear as the child of the *handle* resource, use the *Selected Object Properties* dialog to set the *HasResources* flag of the *handle* polygon to YES.

12. Check the resource browser. You can see there the *$Widget* at the top level, then the *handle* resource below it, and the *handle_width* below that.

13. Select *Run, Start* and at the run prompt, change the default attribute name *LineWidth* to the resource name *handle_width*, and issue the command:
```
$datagen -sin d 0 10 $Widget/handle/handle_width
```

Geometrical Transformation Animation

Now that you have seen animating simple attributes and resources, we will add a geometrical transformation and animate that. Most valve handles turn, so we will add a rotation transformation.

1. If it is still going, stop the animation with *Run, Stop*. Select the viewport, select *Traverse*, *Hierarchy Down*, and select the *handle* polygon.

2. Open the *Selected Object Properties* dialog, and press the *Add Dynamics* button in it (or just click the Add Dynamics toolbar button). This opens the *Add Dynamics* dialog.

3. In the transformation dialog, click on the *Transformation Type* pulldown list, and select "Rotate." The *Rotation Axis* pulldown list appears. Select "Z" from that list to make the rotation happen in the plane of the drawing.

4. Press *Center In Drawing* and notice a prompt at the bottom of the screen. Select a point around which the polygon will rotate. A round red marker with a cross appears at that spot.

5. Set the *Variable Name* field to read "rotate_factor," and press the *Apply* button at the bottom of the dialog. This will attach the transformation and opens the *Edit Dynamics* dialog for editing its parameters. The dialog may later be accessed by using the *Edit Dynamics* button of the *Properties* dialog, or using the *Edit Dynamics* toolbar button.

6. The attributes of the rotation transformation are displayed in the dialog. The center point around which the object is rotated is there, as well as two other attributes: *Factor* and *Angle*. The angle circumscribed by a rotation transformation is given by the product of these two attributes. The *Factor* attribute is usually used as a normalized value, while the *Angle* is usually the maximum angle, in degrees. You can animate the transformation with either

attribute. Note that if you press the ... button next to the *Factor* attribute name, you can see a dialog that says that this attribute is named *rotate_factor*, the name you typed in the previous step.

7. Use *Traverse, Up* to go to the top level of the drawing, and open the resource browser. You can see that *rotate_factor* is now a resource of the *handle* object, which is a resource of the *$Widget* viewport.

8. Select *Run, Start* and at the run prompt issue the command:
```
$datagen -sin d 0 1 $Widget/handle/rotate_factor
```

9. Stop the animation, select the viewport, *Traverse*, *Hierarchy Down*, select the polygon, bring up the *Selected Object Properties* dialog, and press *Edit Dynamics*.

10. Set *Factor* to 1. Press the ... button next to the *Angle* attribute, and give it the name, "rotate_angle."

11. Select *Run, Start* and at the run prompt issue the command:
```
$datagen -sin d 0 90 $Widget/handle/rotate_angle
```
Notice that the data range is from 0 to 90 now.

Creating Copies and Animating Them

Now we will add a base to the valve handle, and reproduce it so we have two valves.

1. Stop the animation, select the viewport, *Traverse*, *Hierarchy Down*, and draw another polygon to represent the base of a valve.

2. Select *Arrange, Group*. Click in the drawing and drag the cursor to display a box. Any objects within or touching the box will be placed into the new group. Use this to group the *handle* polygon, and the new polygon for the base. Display the *Selected Object Properties* dialog for the group, name it "valve1," and set its *HasResources* flag to YES.

3. Set the *valve1* group's *MoveMode* to STICKY MODE. This setting is important when the object is moved by dragging it with the mouse. When the valve group's *MoveMode* is set to MOVE POINTS, moving the group moves all objects in the group by changing coordinates of their points, but it does not move the center point of the rotation transformation together with the rest of the objects. This means that after the valve is moved, it will still rotate around the original point in the drawing. If *MoveMode* is set to STICKY MODE, the center of rotation will move with the object, and the handle will rotate around the same position relatively to the valve.

4. Create a copy of the *valve1* group. You can use *Edit, Cut* and *Edit ,Paste*, or just *Edit, Full Clone*. Move the copy somewhere that doesn't obscure the original.

5. Use the *Selected Object Properties* dialog to rename the new group "valve2."

6. Animate the valve (use *Run, Start*) with the command:
```
$datagen -sin d 0 90
   $Widget/valve1/handle/rotate_angle
```

7. Stop the animation and try it again with:
```
$datagen -sin d 0 90
   $Widget/valve2/handle/rotate_angle
```

8. Try it again with:
```
$datagen
 -sin d 0 90 $Widget/valve1/handle/rotate_angle
 -sin d 0 90 $Widget/valve2/handle/rotate_angle
```

9. Create an ordinary text file called "valve" containing:
```
 -sin d 0 90 $Widget/valve1/handle/rotate_angle
 -sin d 0 90 $Widget/valve2/handle/rotate_angle
```
Now animate the valve with the command:
```
$datagen -argf valve
```

10. To use this drawing in a program, you would use "valve1/handle/rotate_angle" and "valve2/handle/rotate_angle" as input resource names for *GlgSetDResource* or *GlgGetDResource*.

Constraining Attributes

Here we will add a constraint to the rotation of the two valves, so they will always rotate together. Constraints like these are the heart of the GLG drawing architecture.

1. Select the viewport, *Traverse*, *Hierarchy Down*, select the *valve2* group, *Traverse*, *Hierarchy Down*, select the *handle* polygon, display the *Selected Object Properties* dialog, and press the *Edit Dynamics* button.

2. Press the [...] button next to the *Angle* attribute, and press the *Constrain* button on the left side of the *Attribute* dialog.

3. Press the *Use Resource* button in the *Attribute* dialog, then use the resource browser to select the resource: *$Widget/valve1/handle/rotate_angle*.

4. Animate the valve with the command:
```
$datagen -sin d 0 90
   $Widget/valve1/handle/rotate_angle
```
Notice that both valves move together.

This example is only meant to illustrate some of the basic procedures involved in using the GLG Graphics Builder to create and animate a GLG drawing, and it only scratches the surface of what is possible. You may find it profitable to work out some similar simple exercises before starting in on a large project.

Builder Setup and Customization

Environment Variables

Environment variables may be used to define the location of the GLG installation directory and other GLG components. All environment variables have two forms: a generic form (i.e. GLG_DIR) and a version-specific form (i.e. GLG_DIR_X_Y, where X and Y are the major and minor GLG version numbers). The version specific form may be used to prevent conflicts when one machine has more than one version of the GLG Builder installed. The following environment variables are supported:

GLG_DIR

Defines the location of the GLG installation directory, is set by default on Windows.

GLG_CONFIG_FILE

GLG_HMI_CONFIG_FILE

Defines the location of the configuration files for the Graphics Builder and HMI Configurator, if it is different from the default.

GLG_PALETTES_LOCATION

GLG_HMI_PALETTES_LOCATION

Defines the location of the GLG widgets directory for the Graphics Builder and HMI Configurator, if it is different from the default.

GLG_LOG_DIR

Defines the directory of the GLG error log file, if it is different from the default.

GLM_LOG_DIR

Defines the directory of the error log file for the Builder's Map Server component, if it is different from the default.

GLG_VERBOSE

If set to *True*, enables additional output when troubleshooting OpenGL driver, editor setup, loadable editor extension DLLs and other editor extensions. The *-verbose* command-line option may also be used.

GLG_OPENGL_MODE

If set to *True*, enables the OpenGL renderer. If set to *False*, a native windowing system renderer will be used. The *-glg-enable-opengl* and *-glg-disable-opengl* command-line options may also be used.

GLG_OPENGL_VERSION

Specifies the value of *GlgOpenGLVersion* which requests the specified OpenGL version. The shader-based Core OpenGL profile is used for OpenGL versions higher than 3.00, and the Compatibility profile is used for older OpenGL versions. The Compatibility profile is used by default; an OpenGL version needs to be explicitly specified to use the Core profile.

GLG_OPENGL_HARDWARE_THRESHOLD

Specifies the value of *GlgOpenGLHardwareThreshold*. All viewports with a non-zero value of the *OpenGLHint* attribute, less than or equal to *GlgOpenGLHardwareThreshold*, will be rendered using the hardware OpenGL renderer (if available). Viewports with the attribute value between *GlgOpenGLHardwareThreshold* and *GlgOpenGLThreshold* will be rendered using the software OpenGL renderer (if available).

GLG_OPENGL_THRESHOLD

Specifies the value of *GlgOpenGLThreshold*. All viewports with a value of the *OpenGLHint* attribute greater than *GlgOpenGLThreshold* will be rendered using the GDI renderer.
`-glg-disable-hardware-opengl`

GLG_DISABLE_HARDWARE_OPENGL

If set to True, disables hardware OpenGL. If the OpenGL driver is enabled, only the software-based OpenGL renderer will be used.

GLG_DISABLE_SOFTWARE_OPENGL

Disables software OpenGL. If the OpenGL driver is enabled, only the hardware-based OpenGL renderer will be used.

GLG_DEBUG_OPENGL

If set to True, generates extended diagnostic output for the OpenGL driver.

GLG_DISABLE_TIMERS

Disables timer transformations in the drawings for debugging purposes.

GLG_WIDGET_EDITING_MODE

If set to True, widgets loaded from the palettes with *Ctrl-click* may be saved into the original drawing files, facilitating convenient editing of widgets in the custom widget palettes. Without this option, a copy of a modified widget is saved in the current directory by default, to avoid permanently overwriting widgets in the GLG Builder palettes.

Note: Environment variables that control diagnostic output and driver rendering modes modify the behavior of both the GLG Builder and the GLG applications at run-time. To modify only the Builder's behavior, use the corresponding global environment variables settings in the Builder configuration file, if possible. Refer to the *Appendices* chapter on page 341 of the *GLG Programing Reference Manual* for a full list of global configuration resources and environment variables.

Builder Configuration File

The *glg_config* configuration file contains the most common initial settings for customizing the GLG Builder, such as a number of colors in the color palette, color RGB entries format, modality of the Builder's dialogs and other options. The configuration file is located in the GLG installation directory by default. On Unix, it may be named "*.glg_config"* and placed into the users' home

directory to allow per-user customization. The GLG_CONFIG_FILE environment variable described in the previous chapter may be set to point to a configuration file in a non-standard location. The Builder configuration file affects only the GLG Graphics Builder. It does not affect GLG applications at run time, which must use GLG Programming API for its customization.

To avoid conflicts between several versions of the GLG Toolkit installed on the same machine, a version specific configuration file may be provided in the form *glg_config_X_Y*, where X and Y are the major and minor GLG version numbers.

For the HMI Configurator, the *glg_hmi_config* configuration file and GLG_HMI_CONFIG_FILE environment variable are used.

Configuration File Syntax

Each line of the configuration file contains a name of the configuration variable followed by the "=" sign and the variable's value. A string is expected as a value for variables of the S type, a double value must be specified for variables of the D type, and a triplet of three double values must be specified for variables of the G type.

The configuration file also provides access to the GLG global configuration resources. The variables in the configuration file whose name starts with the "*Glg*" prefix provide initial setting for the corresponding global configuration resource. For example, the *GlgPickResolution* variable in the configuration file sets the value of the *GlgPickResolution* global configuration resource. Refer to the *Appendices* chapter of the *GLG Programming Reference Manual* for a list of the global configuration resources.

Comment lines may be introduced by using the "#" character at the beginning of the line.

Configuration Variables

Configuration variables for the Graphics Builder and HMI Configurator are described in the self-documented configuration files, *glg_config* and *glg_hmi_config*, respectively.

Custom Widget Palettes

Custom widgets can be added to the existing palettes by editing *.pal* palette files. Custom palettes can also be integrated into the GLG Builder and HMI Configurator by adding them the *palettes.pls* file, refer to the *Adding Custom Widgets and Custom Palettes* chapter on page 303 for more information.

OEM Customization

OEM Customization features allow extending GLG editors' functionality with application-specific features. It may be applied to both the GLG Graphics Builder and GLG HMI Configurator.

The simplest customization includes defining a custom color palette and custom dynamics options for the OEM version of the GLG editor. For more elaborate customization, the OEM integrator may provide custom components with a predefined set of properties for use with the GLG HMI

Configurator. For even further customization, loadable DLLs may be provided to extend GLG editors with custom OEM functionality, such as connecting to a custom data source or supplying application-specific menu options, toolbar icons and dialogs.

Custom Color Palette

In addition to the default color palette, GLG editors provide a custom color palette. The custom color palette may be activated by using *Options, Color Options, Swap Color Palettes*. If the color palette is displayed, *Ctrl+click* on it also toggles the displayed palette.

To supply your own custom color palette, edit the custom color palette template drawing provided in the *editor_extensions/drawings/custom_color_palette.g* directory of the GLG installation and copy it to the main directory of the GLG installation. The *-custom-color-palette* command-line option may also be used to supply a custom color palette drawing for the Builder or HMI configurator:

-custom-color-palette editor_extensions/drawings/custom_color_palette.g

The custom color palette drawing can also be specified in the GLG configuration file using the *CustomColorPalette* variable, or by setting the GLG_CUSTOM_COLOR_PALETTE environment variable.

The template drawing contains a GLG palette widget with the *GlgPalette* interaction handler. The widget contains two groups: the *PaletteObject* group containing objects whose *FillColor* attribute defines the palette's colors, and an optional *Labels* group that contains text labels used for annotating the color names. To change the palette, add or delete the objects in each group as needed and change their colors. Refer to the *GlgPalette* section on page 212 for details of the *GlgPalette* input handler.

OEM Version of the Graphics Builder

The OEM version of the GLG Graphics Builder provides additional functionality for defining and browsing public properties. Since this functionality is used only for OEM customization, it is provided only when requested via a command-line option to avoid cluttering user interface for the rest of the users. The OEM version may be started by using the *-oem* command-line option of the Enterprise Edition of the Graphics Builder:

```
GlgBuilder  -oem
```

Export Tags

Export tag is a special type of a tag object that can be attached to an object's attribute to mark it as an exported **public property**. Public properties of an object may be displayed using the *Object, Public Properties* menu option of the OEM version of the Graphics Builder. Public properties are also displayed in the GLG HMI Configurator, which allows creating **custom components** with user-defined properties for use with the HMI Configurator.

Instead of a single button to add data tags, the OEM version of the Builder provides two buttons for adding tags: the *DT* button for adding Data Tags and *ET* for adding Export Tags. If a data or export tag has been added to the attribute, the corresponding button's label changes to *DT*+ or *ET*+ to indicate the presence of a tag. If a tag is present, it may be edited by clicking on the corresponding button.

The export tag's *TagName* attribute defines the name of the public property. This name will be displayed as a property name for the attribute in the *Public Properties* dialog. The export tag's *TagType* attribute may have the following values:

EXPORT
: Used to mark the attribute as a public property.

EXPORT_DYN
: Used to mark the attribute as a public property of predefined dynamics, action commands and action data sets. Refer to the *Custom Predefined Dynamics* section below for more information.

When accessing attributes of an object via resources, the export tags may be accessed only for resources that use default attribute names and not for named resources.

Public properties are global: all public properties appear in the object's public properties list regardless of the hierarchy level they are defined at. The *GlgQueryTags* and other GLG API methods may be invoked with the *tag_type* parameter set to GLG_EXPORT_TAG to query object's public properties.

Public Properties

Public properties defined in the OEM version of the Builder are used by the HMI Configurator. If an object has public properties defined via export tags, the object may be used in the GLG HMI Configurator as a **custom component**.

The HMI Configurator has several options (defined in the *glg_hmi_config* configuration file) for editing custom components with user-defined public properties. The HMI Configurator's *Property* button may be configured to display public properties for custom components, and attributes for objects that do not have public properties. The HMI Configurator may also be configured to provide two *Property* buttons: one for displaying object's public properties and another for displaying its attributes.

Public properties provide a mechanism for supplying custom components with user-defined properties for editing in the HMI Configurator. The HMI Configurator's configuration file has a number of options to restrict editing of custom components to editing just their exported public properties.

The OEM version of the Graphics Builder provides two menu choices and two toolbar buttons for displaying object properties: one displays the object's attributes and another displays its public properties. The non-OEM version of the Builder provides both *Properties* and *Public Properties* menu options, but only one *Property* toolbar button.

The *Palettes, HMI Editor Widget Samples* menu displays a palette with samples of HMI components that have public properties. The Builder's *Object, Public Properties* menu option may be used to browse custom properties of a selected component.

Custom Components with User-Defined Properties

Custom components are objects with public properties defined using export tags described in the previous chapter. Custom components with user-defined properties are used with the GLG HMI Configurator to provide end users with application-specific building blocks with properties that are related to the application logic. Refer to the previous chapter for more information on using custom components in the HMI Configurator.

To create a custom component for use with the GLG HMI Configurator:

1. Start the OEM version of the Graphics Builder by using the *-oem* command-line option of the Enterprise Edition of the Graphics Builder.

2. Create graphics to represent the component. If the graphics contains several objects, use a group or a container object to encapsulate the graphics as one object. Add any required dynamics.

3. Add export tags to the attributes of the graphics or dynamics to define public properties.

4. Name the top level object *$Drawing* to use it in the HMI Configurator's palettes and provide an optional *$Icon* graphics if desired. Refer to the *Palettes* chapter on page 302 for more information on palette drawing conventions.

5. Save the drawing and copy it into one of the HMI Configurator's palettes directories. Refer to the *Adding Custom Widgets and Custom Palettes* chapter on page 303 for more information.

Objects created in the HMI Configurator are marked with an internal *HMIFlag* flag to differentiate them from the custom components and objects created in the Graphics Builder. The HMI Configurator imposes restrictions on editing objects created in the Graphics Builder. For example, the HMI Configurator allows to explode group objects only if they were created in the HMI Configurator. The *Options, OEM Options, Toggle Object's HMI Flag* option may be used to change *HMIFlag* setting if required.

Custom Predefined Dynamics

GLG editors provide two sets of dynamics options for object attributes: the **stock transformations** and **predefined dynamics**. The stock transformations are basic transformation types used as building blocks to implement dynamic behavior. Predefined dynamics provide easy to use options for the most common dynamic actions in the GLG editors. Predefined dynamics may also be used by system integrators to extend GLG editors with elaborate application-specific dynamics.

Predefined dynamics are implemented using custom transformations which represent a collection of several stock transformations wired together to implement specific dynamic behavior. Most of the parameters of the transformations used to implement the predefined dynamics are hidden from

the user, and only the essential parameters marked as public are presented to the user as a simple list of public properties. The *Options, Dynamics Options* menu of the Graphics Builder contains options that control how the predefined dynamics are displayed in the Builder's dialogs.

Predefined Dynamics Template Drawing

The *editor_extensions/drawings/custom_xform_templates.g* file of the GLG installation provides a template that contains the default predefined dynamics. To add custom predefined dynamics, this drawing may be edited using the OEM version of the Enterprise Edition of the Builder (use the *-oem* command-line option to start the Builder in the OEM mode).

When finished, copy the drawing to the main directory of the GLG installation. The *-xform-templates* command-line option may also be used to supply the drawing containing a custom predefined dynamics template for the Builder or HMI configurator:

-xform-templates <glg_dir>/editor_extensions/drawings/custom_xform_templates.g

The drawing containing a custom predefined dynamics template can also be specified in the GLG configuration file using the *CustomXformTemplates* variable, or by setting the GLG_CUSTOM_XFORM_TEMPLATES environment variable.

The predefined dynamics template drawing contains the *XformTemplates* viewport, which contains several groups of objects, one for each transformation subtype:

- *DXform* group defines transformations for generic D attributes;
- *SXform* group defines transformations for generic S attributes;
- *GXform* group defines transformations for generic G attributes;
- *ColorXform* group defines transformations for color attributes.
- *LineTypeXform* group defines additional transformations for *LineType* attributes. These transformations will be available for *LineType* attributes in addition to the predefined transformation defined in the *DXform* group.
- *VisibilityXform* group defines transformations for object *Visibility* attributes.
- *GeomXform* group defines transformations for geometrical transformations of objects and their control points. It is used only by the GLG HMI Configurator.

All groups are optional and some of the groups can be removed if necessary. Each object in the group defines a certain type of predefined dynamics, and the name of the object in the group defines the label displayed in the predefined dynamics menu. The order of predefined dynamics in the menu is defined by the drawing order of the objects in each group, not by the visual order of their appearance in the drawing. The *Arrange, Reorder* menu options may be used to change the drawing order of the objects in a group.

Each object in a group has a transformation named *XformObject* attached to one of the object attributes, as indicated by the "*X*" button on the right side of an attribute row in the object's *Properties* dialog. For objects in the *GeomXform* group, the transformations are added to the objects

themselves instead of their attributes. The transformation defines custom predefined dynamics and may contain a complex collection of transformations wired together to perform a custom dynamic action.

The object must also define an S type resource named *XformLabel* which defines a custom transformation type shown in the *Edit Dynamics* dialog. The *XformLabel* resource is usually defined as a custom object property accessible via the *Data* button in the status panel.

Some transformation (such as *Flow* in the *LineTypeXform* group) may have *InitFromObject* custom property attached to the object. Setting the value of *InitFromObject* to 0 ensures that the transformation is not modified when it is attached to an object. By default, the first element of a list transformation is initialized to the value of the attribute the list transformation is attached to. This preserves the color of the object when a list transformation is attached to its color attributes, or preserves the text string when a list is attached to the TextString attribute. For some transformations, such as *Flow*, such initialization would interfere with the transformation's logic, and setting *InitFromObject* to 0 prevents the transformation from being modified when it is attached to an attribute.

The *XformObject* transformation uses export tags to define public properties visible in the *Edit Dynamics* dialog. Export tags are attached to the attributes of the transformation that need to be visible to the user. The export tag's *TagName* defines the property name that will be shown in the *Edit Dynamics* dialog. The export tag's *TagType* must be set to EXPORT_DYN to export the attribute as a public property of the predefined dynamics. Refer to the *OEM Version of the Graphics Builder* section above for more information on export tags.

Once export tags are added to a transformation, its *Edit Dynamic* dialog will list the transformation's exported public properties instead of its attributes. The *Options, Dynamics Options, Full Display of Predefined Dynamics* menu option toggles the display of predefined dynamics between public properties and attributes. This option can be used for getting access to the attributes of the transformation for editing.

To allow accessing the attribute via resources in the HMI Configurator, the attribute is usually named the same way as the public property name specified by the *TagName* attribute of the export tag. Data tags may be added to the exported public properties for use with the HMI Configurator. The name of a data tag is specified via the data tag's *TagName* attribute and usually matches the *TagName* of the attribute's export tag.

Predefined dynamics are usually constructed by wiring together several transformations, with second-level transformations added to the attributes of the top-level transformation. Public properties are global: public properties defined by export tags of transformations on any level are listed as a flat list of public properties for the top-level transformation.

If predefined dynamics are added to a transformation's attribute as a second-level transformation instead of a stock transformation, public properties of the second-level transformation are not listed as the properties of the parent transformation due to the setting of an internal flag. To unset the flag and make public properties of the second-level transformation listed as properties of its parent, uncheck the *Options, OEM Options, Toggle Custom Xform Flag* option for the second-level transformation object.

The *LineTypeXform* group defines additional transformation for the *LineType* attribute, in addition to the transformations from the *DXform* group that will be automatically appended to the list.

Adding Predefined Dynamics

To add or delete predefined dynamic options, add or delete objects from the corresponding groups in the template drawing and edit the *XformObject* transformations attached to the objects' attributes. To define new predefined dynamics:

1. Start the OEM version of the Graphics Builder by using the *-oem* command-line option of the Enterprise Edition of the Graphics Builder and load the predefined dynamics template drawing from the *<glg_dir>/editor_extensions/drawings/custom_xform_templates.g* file.

2. Add a new object to a corresponding group and define its name and *XformLabel* property as listed above.

3. Make sure that the *Options, Dynamics Options, Full Display of Predefined Dynamics* option is checked to display the transformation's attributes instead of public properties.

4. Add a stock transformation to the object.

5. Add transformations to any of its attributes as required to implement custom dynamic behavior. Use stock transformations to simplify the process of defining public properties.

6. Add export tags to the attributes of a transformation that need to be shown as transformation's properties. To add an export tag to a transformation's attribute, click on the ellipses button next to the attribute in the *Edit Dynamics* dialog, then press the *ET* (Export Tag) button the *Attribute* dialog.

7. Uncheck the *Options, Dynamics Options, Full Display of Predefined Dynamics* option and verify the list of the transformations' public properties.

8. Use the modified drawing with the *-xform-templates* command-line option to test the new predefined dynamics.

Custom Data Sets and Custom Commands

Custom data sets may be used to define predefined sets of data to be added to objects as custom data, action data or action commands. Custom data sets are contained in a group object that holds individual data elements. Data elements of action and command data have EXPORT_DYN export tags attached to export them as public properties visible in the Public Properties dialog for editing action or command data.

Predefined Custom Command Template

The *editor_extensions/drawings/custom_command_templates.g* file of the GLG installation provides a template that contains predefined commands and data sets. To add custom data sets and/or custom commands, this drawing may be edited using the OEM version of the Enterprise Edition of the Builder (use the *-oem* command-line option to start the Builder in the OEM mode).

When finished, copy the drawing to the main directory of the GLG installation. The *-command-templates* command-line option may also be used to supply the drawing containing a custom predefined commands template for the Builder or HMI configurator:

-command-templates <glg_dir>/ditor_extensions/drawings/custom_command_templates.g

The drawing containing a custom commands template can also be specified in the GLG configuration file using the *CustomCommandTemplates* variable, or by setting the GLG_CUSTOM_COMMAND_TEMPLATES environment variable.

The predefined command template drawing contains three groups of objects:

- *Commands* group defines available commands for the SEND_COMMAND actions.
- *EventDataSets* group defines predefined data sets for the SEND_EVENT actions.
- *CustomDataSets* group defines predefined data sets used for adding custom data to objects.

Each object in the group defines a command or a predefined data set, and the name of an object will be used as a label displayed in the list of choices in the Builder. The order of commands or data sets in the list is defined by the drawing order of the objects in each group, not by the visual order of their appearance in the drawing. The *Arrange, Reorder* menu options may be used to change the drawing order of objects in a group.

Each object in a group has a custom data attached to define a corresponding command or data set. The custom data attached to each object define data elements of the corresponding command or data set. Data elements of command or action data sets use export tags to define public properties visible in the Edit Command or Edit Action Data dialog. The export tag's *TagName* defines the property name that will be shown in the edit dialog. The export tag's *TagType* must be set to EXPORT_DYN to export the attribute as a public property of the predefined data set. Refer to the OEM Version of the Graphics Builder section above for more information on export tags.

The *Options, Selection Options, Edit Action Data as List* menu option toggles the way action and command data of the SEND_COMMAND and SEND_EVENT actions are displayed for editing: as a public properties dialog, or a list of properties that allows adding and removing individual properties. A button  in the upper right corner of the Action Properties dialog provides a convenient shortcut.

To allow accessing data elements of each data set via resources, each data element is named. For the command and action data sets, the name is the same as the public property name specified by the *TagName* attribute of the export tag.

Adding Custom Commands and Custom Data Sets

To add or delete commands or custom datasets, add or delete objects from the corresponding group in the command template drawing. To define new commands or custom data sets:

1. Start the OEM version of the Graphics Builder by using the *-oem* command-line option of the Enterprise Edition of the Graphics Builder and load the command template drawing from the *<glg_dir>/editor_extensions/drawings/custom_command_templates.g* file.

2. Add a new object to a corresponding group, define its name and add custom data with data elements as described above.

3. Use the modified drawing with the *-command-templates* command-line option to test the new commands and custom data sets.

4. Uncheck the *Options, Selection Options, Edit Action Data as List* menu option, add new command or custom event data set and verify the list of its properties.

OEM Editor Extensions

GLG editors support OEM editor extensions in the form of loadable DLLs (or shared libraries on Linux/Unix platform). The same DLL may be used for all editions of the GLG Graphics Builder as well as the GLG HMI Configurator.

The following extensions are available and described in the next sections:

- *Custom Data Browser Extension*
- *Custom Run Module Extension*
- *Custom Editor Options and Dialogs Extension.*

The GLG installation includes examples of all available extensions in the *editor_extensions* directory. All examples contain a run script for starting a GLG editor with the extension DLL. They also contain self-documented source code, a makefile and/or project file for building the extension, and a README file with more information. The *Editor Extension API Files* section of this chapter describes common files used by all DLL examples.

All OEM Extension DLLs may use both the GLG Standard and Extended API for implementing the extension. Since the DLLs are used with the GLG editor and use its Extended API, the extension DLLs themselves are royalty-free and do not require any additional GLG libraries.

The custom extension DLLs may be deployed in the Graphics Builder or HMI Configurator by using the command-line options or configuration file variables listed in the corresponding sections below. Each DLL may also be deployed by using a default DLL name and placing the DLL into the directory where the editor executable is located. On **Linux/Unix**, an extension DLL with a default name may also be placed into any location where it will be searched for by the dynamic linker (such as */usr/lib* or a location specified by a LD_LIBRARY_PATH environment variable). On **Windows**, an extension DLL with a default name may also be placed into any location where it will be searched for by the *LoadLibrary* function.

The default name of an extension DLL is formed by a base name and extension. The following lists the default base names of the custom extension DLLs:

Custom Data Browser DLL
libglg_custom_data (Linux/Unix)
glg_custom_data (Windows)

Custom Run Module DLL
libglg_custom_proto (Linux/Unix)
glg_custom_proto (Windows)
Custom Editor Options DLL
libglg_custom_option (Linux/Unix)
glg_custom_option (Windows)

The extension uses a platform's standard extension for dynamic libraries: *.dll* on Windows, *.so* on Linux and most Unix platforms, *.sl* on HPUX. For example, the default name of the Custom Data Browser DLL is *glg_custom_data.dll* on Windows and *libglg_custom_data.so* on Linux.

Custom Data Browser DLL

A custom data browser DLL may be provided for connecting to proprietary data sources to browse available tags in a GLG editor to select a tag's data source. The *Browse* button in the *Data Tag* dialog starts a *Data Browser* that will use the supplied custom data DLL for selecting a tag source. When the tag selection is made, the selected tag is inserted into the data tag's *TagSource* field. The DLL is also used by the Data Browser widget available at the application runtime.

The example is located in the *editor_extensions/data_browser_example* directory and works with both the GLG Graphics Builder and the GLG HMI Configurator.

The *run_data_example* script in the example's directory may be used to run the GLG Builder with a custom data browser. The script can be edited to define the version of the GLG editor to run. To test the data browser, run the script to start the Builder, add a tag to an object's attribute and click on the Browse button of the *Data Tag* dialog to start the tag browser. Select a controller, tag group and tag, then press *Select* to insert selected tag into the *TagSource* field.

The example source code is self-documented and provides an example of browsing a hierarchical process database with several levels of hierarchy: controller, tag group and tag. The syntax used to separate the controller, tag group and tag entries in the selected *TagSource* is just an example and may be changed to fulfill custom application requirements.

A custom data browser DLL can be deployed via the *-data-lib* command-line option as shown in the README file, via the *CustomDataLib* variable of the *glg_config* and *glg_hmi_config* configuration files, or by placing a DLL with a default name in the directory of the GLG Editor or application program executable.

Custom Run Module DLL

A custom proto DLL may be provided for animating the drawing with real data in the Run mode of the GLG editor, as well as handling user interaction. object selection and custom runtime dialogs with an application-specific runtime logic.

The module has access to a complete GLG API, both the Standard and Extended, making it possible to implement a complete application integrated with a GLG editor. The application will function in the editor's Run mode, while the Edit mode may be used for editing the application's drawing.

For even further customization, the *-run* command-line option or the *StartRun* configuration file variable can be used to start the GLG editor in the Run mode. The *-run-window* command-line option or the *RunWindow* configuration file variable can be used to start the Run mode in a separate window, hiding the GLG editor's menus and toolbars. The custom option DLL described in the next section may be used to add custom OEM menu options for the Run mode.

Since the module uses the GLG API supplied by the GLG editor's executable, it may use both the GLG Standard and the Extended API with no additional GLG libraries required. When the module is used with the GLG Graphics Builder or HMI Configurator, the editors provide the module with the Run-Time license for the Extended API.

The example is located in the *editor_extensions/custom_proto_example* directory and works with both the GLG Graphics Builder and the GLG HMI Configurator.

The *run_proto_example* script in the example's directory may be used to run the GLG Builder with a custom data browser. The script can be edited to define the version of the GLG editor to run.

The sample source code is self-documented and provides an example of animating objects in the drawing via tags. The code queries a list of all tags defined in the drawing and animates them with random data. In a real application, the code can animate the tags with real data from a process database based on the tags' *TagSource*. Tags based data access allows an application to animate an arbitrary drawing without any knowledge about its structure or resources. The example also demonstrates the use of resources, custom popup dialogs and handling user interaction.

To check the proto DLL's functionality, run the *run_proto_example* script to start the Builder, then start the Run mode. The DLL will load and display the popup dialog from the *dialog.g* file, updating it with the status information using resources. The DLL receives and processes all input events. When the user presses the *Stop* button, the DLL stops the Run mode of the Builder and returns to the Edit mode.

A custom proto DLL can be deployed via the *-proto-lib* command-line option as shown in the README file, via the *CustomProtoLib* variable of the *glg_config* and *glg_hmi_config* configuration files, or by placing a DLL with a default name in the directory of the GLG Editor executable.

In addition to the cross-platform GLG-based dialogs, the module may also use native dialogs, based on Windows' Win32 API or Xt/Motif on Linux/Unix.

Custom Editor Options and Dialogs DLL

The custom editor options DLL may be provided for adding custom icons, menu options and dialogs with application-specific logic to the GLG editors. The module may also be used to verify the drawing against a custom set of rules before saving it into a file, as well as remove unwanted editor icons and menu options.

The example is located in the *editor_extensions/data_browser_example* directory and works with both the GLG Graphics Builder and the GLG HMI Configurator.

The *run_data_example* script in the example's directory may be used to run the GLG Builder with a custom data browser. The script can be edited to define the version of the GLG editor to run.

The sample source code is self-documented and demonstrates how to add a custom OEM menus and toolbar icons to a GLG editor. The code provides examples of implementing both push buttons and toggle buttons, as well as cascading sub-menus. It also shows how to change sensitivity of the menu options depending on an object selection and the GLG editor's mode: Edit or Run.

One of the OEM menu options demonstrates how to implement a custom OEM dialog that performs a custom OEM action in the editor's Edit mode. The example also includes code samples showing how to customize the GLG editor by removing unwanted icons and menu options.

To check the proto DLL's functionality, run the *run_proto_example* script to start the Builder and notice the *OEM Sample Menu* appearing after the *Edit* menu. It will also appears in the popup menu. A custom OEM icon (a red square with the OEM label) will appear near the right side of the editor's toolbar. The OEM icon and menu entries become active when an object is selected.

Create an object, select it and try the OEM menu options. The *Add/Edit Custom Value* menu option and the *OEM* toolbar icon activate an OEM dialog that adds an OEM property to the selected object and allows the user to edit its value. The property is visible in the Resource Browser as *OEMProperty*. The sample code also checks and sets the object's *HasResources* flag if necessary. The code demonstrates how to implement both modal and non-modal custom dialogs.

The OEM menu contains options for both the Edit and Run modes. Starting the Run mode disables edit options and enables runtime options of the OEM menu. The custom editor options DLL may also implement the functionality of the custom run mode DLL described in the previous section, making it possible to provide a single DLL that handles both the OEM editor options and the OEM runtime mode.

A custom options DLL can be deployed via the *-option-lib* command-line option as shown in the README file, via the *CustomOptionLib* variable of the *glg_config* and *glg_hmi_config* configuration files, or by placing a DLL with a default name in the directory of the GLG Editor executable.

Refer to the *ADDING CUSTOM ICONS* section of the README file for information on the process of defining custom icons used by the module.

In addition to the cross-platform GLG-based dialogs, the module may also use native dialogs, based on Windows' Win32 API or Xt/Motif on Linux/Unix.

Editor Extension API Files

All extension DLL examples project files (and makefiles on Linux/Unix) for building the extension DLL. The following files are provided in the example directories:

sample.c
: Example's source code.

glg_custom_dll.o
: Provides GLG API for the custom DLL. Don't delete this file when cleaning the project directory.

glg_custom_dll.h
: An include file for using the GLG API in a custom DLL.

The *Custom Run Mode* and *Custom Editor Options* examples also provide the following files:

glg_custom_editor_dll.o
: Provides GLG Editor Extension API for the custom DLL. Don't delete this file when cleaning up the project directory.

glg_custom_editor_dll.h
: An include file for using the GLG Editor Extension API in a custom DLL.

Chapter 7

GLG Graphics Builder Menus

7

This chapter presents descriptions of all the options of the GLG Graphics Builder's menus. They are organized by menu name, in the same order as they appear in the Builder's menu bar:

- The *File* Menu provides options that mainly deal with saving and printing drawing files, saving images and various exported files. These options are described starting on page 295.
- The *Palettes* Menu provides access to palettes of widgets and other pre-built objects. These options are described starting on page 302.
- The *Edit* Menu provides options for selecting, editing and copying objects, as well as undo options. These options are described starting on page 305.
- The *View* Menu provides options for controlling your view of the drawing. These options are described starting on page 310.
- The *Traverse* Menu provides options for working with advanced objects. These options are described starting on page 313.
- The *Arrange* Menu provides options for working with groups, as well as other miscellaneous options described starting on page 317.
- The *Layout* menu provides options for aligning and positioning objects.
- The *Object* Menu provides options for creating and manipulating objects. These options are described starting on page 326.
- The *Run* Menu provides options that let you animate a drawing. These options are described starting on page 354.
- The *Options* Menu provides options that control the appearance and function of the GLG Graphics Builder. These options are described starting on page 356.

This chapter describes all menu options available in the Enterprise Edition of the Graphics Builder. Some options are not present in the Basic and Professional Editions.

The toolbar below the menu bar provides convenient shortcuts for accessing the most often used menu choices. To see the tooltip showing the function of a toolbar's button, hold the mouse inside the button until the tooltip appears.

File

The *File* menu provides options to let you load and save drawing files, export/import strings and tags, print drawings, generate images, and close the GLG Graphics Builder.

New

The *New* submenu provides options to create widgets and subdrawings with various resize policies. The entries with the *Resizable* option use the world coordinate system; they stretch or change the size of objects in the drawing accordingly when the drawing is resized. The entries with the *Fixed*

Scale option use the screen coordinate system and objects in the drawing do not change their size when the drawing is resized. Instead, more or less of the drawing area is shown when the window is resized. If the grid is ON, the grid interval is adjusted to match the selected *Resizable* or *Fixed Scale* option.

Widget (Resizable)

New, Widget (Resizable) starts a new resizable drawing by creating a new widget and placing editing focus into it. The widget's resize policy resizes all objects in the drawing when the widget is resized.

If the drawing area already contains objects, they are not saved. The Builder asks if you want to discard the current drawing, but it does not prompt you to save any changes to the current drawing. You must explicitly save the drawing using *File*, *Save*.

A new drawing area contains two objects: *MMDrawingArea* and *MMAxisIcon*, which are visible in the Resource Browser and Properties Dialog of the drawing area. *MMDrawingArea* represents the drawing area itself, and *MMAxisIcon* displays the three axes to shows the orientation of the view. *MMAxisIcon* is only displayed when the drawing area's *Axis* attribute is turned on. These objects are not part of the drawing, but part of the Builder.

To create objects, use the buttons in the object palette or the choices in the *Object*, *Create* submenu. To use pre-built objects, use the *Palette* menu to load the palettes. When the drawing is saved, the Builder will bring the editing focus back to the top level of the hierarchy, showing the *$Widget* viewport it created for the new widget.

Widget (Fixed Scale)

Same as *New, Widget (Resizable)*, but creates a fixed scale widget. When the widget is resized, its drawing area shows more or less without changing the size of the objects drawn in it. Since the drawing is not stretched, the objects in the drawing always keep their X/Y ratio.

SubDrawing (Resizable)

New, SubDrawing (Resizable) is used to create a resizable drawing which is later used as a template of a resizable subdrawing object.

SubDrawing (Fixed Scale)

New, SubDrawing (Fixed Scale) is used to create a non-resizable drawing to be used as a template of a fixed-size subdrawing object.

Empty Drawing (Resizable)

New, Empty Drawing (Resizable) creates a new resizable drawing without creating a widget. When the drawing is resized, all objects in the drawing are resized as well.

Before drawing objects, we recommend that you create a viewport using *Object*, *Create*, *Viewport*, open it using *Traverse*, *Hierarchy Down*, or use the *New Widget* option to create a new viewport automatically.

To create objects, use the buttons in the object palette or the choices in the *Object*, *Create* submenu. To use pre-built objects, use the *Palettes* menu to load the palettes. *Ctrl*-clicking on the palette's icons loads the corresponding widget as a new drawing.

Empty Drawing (Fixed Scale)

New, Empty Drawing (Fixed Scale) is the same as *New, Empty Drawing (Resizable)*, but creates a non-resizable drawing where objects do not change their size when the drawing is resized. Instead, more or less of the drawing area is shown in the window. Since the drawing is not stretched, the objects in the drawing always keep their X/Y ratio.

Reset Drawing

Reset Drawing initializes the current drawing. The Builder rebuilds each object in the drawing, updating the Builder's representation to match the current information in the object hierarchy.

For composite objects such as series, references, polylines and polysurfaces, *Reset Drawing* ensures that the instance objects reflect any changes to the template. This means that any changes made to series instances are lost when the drawing is reset.

All attribute values of instances are also lost when the drawing is reset. The old instances are destroyed and a new set is replicated with values from the template.

Open

Open loads a drawing from a file.

When you select *Open*, the Builder prompts you to select a file name, using the standard file selection dialog. The Builder can open a drawing saved in any of its own formats; see page 357.

If a drawing is already open, *Open* discards the current drawing. The Builder asks if you want to discard the current drawing, but it does not prompt you to save the changes.

To load more than one drawing into the drawing area, use *File*, *Load Object*; see page 298.

Open URL

Open URL loads a drawing from a URL. In the Unix environment, the GLG_WGET_PATH environment variable must be set to point to the *wget* utility executable to enable this option.

Recent Drawings

Recent Drawings displays a list of the recently edited drawings. Select one of the recent drawings from the list to load it.

If a drawing is already open, loading a recent drawing discards the current drawing. The Builder asks if you want to discard the current drawing, but it does not prompt you to save the changes.

Save

Save saves the current drawing to a file.

The first time you save a drawing, the Builder prompts you for a file name, using the standard file selection dialog. If the drawing has been saved before, it asks if you want to overwrite the existing file.

The Builder can save files in three different formats:

- Binary, which loads quickly but is not portable across platforms that use different binary data representations.
- ASCII, which is completely portable across platforms, but loads more slowly than binary.
- Extended, which is portable across platforms and across versions of the Builder, but loads most slowly.

There is also an option for saving compressed drawings. Compressed drawings are smaller but load slower.

The default format for saving drawings is compressed ASCII. To change the format, use *Options*, *Save Format* and *Save Compressed*; see page 357.

SaveAs

Save As saves the current drawing to a different file.

When you select *Save As*, the Builder prompts you to enter a file name, using the standard file selection dialog. The usual file name extension for a GLG drawing is *.g* though it is not required.

The default format for saving drawings is compressed ASCII. To change the format, use *Options*, *Save Format* and *Save Compressed*; see page 357.

Load Object

Load Object loads an object from a file into an existing drawing. The object is loaded into the current place in the object hierarchy. If the *editing focus* is inside a viewport or a group, the loaded object will be added to that viewport or group.

When you select *Load Object*, the Builder prompts you to select a file name, using the standard file selection dialog. The Builder can open any drawing saved in any GLG format. If the input drawing contains more than one object, they appear together in a newly created group in the existing drawing.

To clear the drawing area before loading a drawing, use *File*, *Open*; see page 297.

To save a drawing to a file, use either *File*, *Save* or *File*, *Save Object*; see page 298.

Recent Objects

Recent Objects displays a list of the recently edited objects. Select one of the recently accessed objects from the list to load it into an existing drawing. Similar to *Load Object*, the object is loaded into the current place in the object hierarchy.

Save Object

Save Object saves the currently selected object to a file. *Save Object* differs from *File*, *Save* because it lets you save a selected part of a drawing. For example, by including objects in a group and then using *Save Object* on the group object, you can isolate part of an object hierarchy.

When you select an object and then select *Save Object*, the Builder prompts you to enter a file name, using the standard file selection dialog. The usual file name extension for a GLG drawing is *.g* though it is not required.

To add a saved object into an existing drawing, use *File*, *Load Object*; see page 298. To edit the saved object, use *File*, *Open* to open the saved object as a separate drawing file; see page 297.

The object is saved using the same format as for *File*, *Save*; to change the format for saving drawings and objects, use *Options*, *Save Format*; see page 357.

Print

For Linux/UNIX users, *Print* saves a PostScript image of the current drawing into a file.

For Windows users, *Print* sends the current drawing to the printer, using the standard Windows print facilities. Use *Export PostScript* to save a PostScript image of the drawing to a file. The *Print* toolbar button can be configured to perform either task using the *ToolbarPrint* configuration file variable.

Print uses the print configuration set by the *File*, *Print Configuration* options. If editing focus is set, it prints the focus viewport instead of the whole drawing.

Export PostScript

Export PostScript (Windows only) saves a PostScript image of the current drawing (in its current state) to a file. On Unix/Linux, the *Print* option performs the same task.

Export PostScript uses the print configuration set by the *File*, *Print Configuration* options. If editing focus is set, it generates PostScript for the focus viewport instead of the whole drawing.

Print Configuration

The *Print Configuration* submenu provides options to let you set up the printer.

Page Layout

Page Layout specifies how to map the drawing to the printed page.

Page Layout presents a viewport that corresponds to the printed area. The position and size of the viewport relatively to the *Drawing Area* define the position and size of the area in which the drawing will be printed relatively to the page.

Resize the page layout viewport with the mouse to define the printing area. Delete the viewport when you have finished.

This option applies to both PostScript and Windows printing.

Stretch

Stretch prints the drawing using the full area of the page. The drawing is scaled to fill the printing area, so the proportions of the printed drawing may not correspond to the drawing's actual proportions. Using the appropriate orientation (portrait or landscape) can help reduce distortion; to preserve the drawing ratio, turn *Stretch* off.

This option applies to both PostScript and Windows printing.

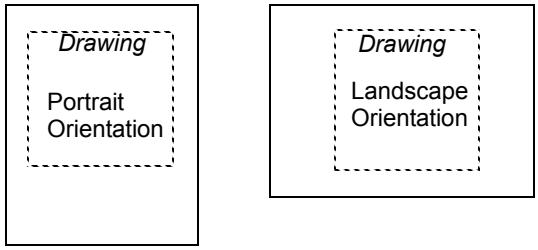
PostScript Level

For PostScript printing and export, *PostScript Level* specifies which version of PostScript the Builder sends to the printer. Level 3 is required for proper PostScript printing of images with transparent background.

PostScript Orientation

For PostScript printing and export, *PostScript Orientation* determines whether the drawing is printed in portrait mode (across the width of the page) or in landscape mode (down the height of the page).

For Windows printing, the page orientation is set in the *Print* dialog.



Portrait and Landscape Orientations

Save Image

Save Image saves the image of the visible part of the drawing area into a file in either the JPEG or PNG format. The format of the image is defined by the *SaveImageFormat* variable in the *glg_config* file. If the editing focus has been moved into a viewport, the image of that viewport will be saved instead of the main drawing area.

Save Image Full

Save Image Full saves an unclipped image of the whole drawing into a file in either the JPEG or PNG format. The format of the image is defined by the *SaveImageFormat* variable in the *glg_config* file. If the editing focus has been moved into a viewport, the image of that viewport will be saved instead of the main drawing area.

Save Direct OpenGL Image

Save Direct OpenGL Image (Windows only) saves a visible part of the drawing into a file by taking an OpenGL snapshot of an image displayed in a viewport. This technique improves the rendering quality of the generated image by getting around the Windows' OpenGL driver, which uses a software renderer with poor anti-aliasing for off-screen rendering.

This option may be used only for viewports with the OpenGL rendering, and it does not work with children viewports.

Export Strings

Export Strings exports all strings defined in the drawing into a file. Refer to the *Localization Support* chapter on page 58 for information about the string translation file format.

Import Strings

Import Strings imports strings from a strings file, replacing matching strings in the drawing. Refer to the *Localization Support* chapter on page 58 for information about the string translation file format.

Export Tags

Export Tags exports all tag names defined in the drawing into a file. Refer to the *Tag Export and Import Features for Run-Time Tag Mapping* chapter on page 61 for information about the tag file format.

Import Tags

Import Tags imports tag names from a tag translation file, replacing matching tag names in the drawing. Refer to the *Tag Export and Import Features for Run-Time Tag Mapping* chapter on page 61 for information about the tag file format.

Exit

Exit closes the GLG Graphics Builder.

If a drawing is already open, the Builder asks if you want to discard the current drawing, but it does not prompt you to save the changes. You must explicitly save the drawing using *File*, *Save* before exiting the GLG Graphics Builder.

Palettes

The *Palettes* Menu provides access to palettes of widgets and other pre-built objects. By default, only the *Custom Objects* and *HMI Editor Widget Samples* palettes are installed. Other palettes are optional and will be installed only if selected. Possible palettes include Real-Time Charts, 2D Graphs, 3D Graphs, Controls, Avionics, Process Control and Special Widgets palettes.

The *Palettes* Menu lists all available palettes of pre-built objects. To display a palette, select it from the palettes list. To add an object from a palette into the drawing, click on its icon in the palette. The Builder will insert a copy of the object into the drawing. Give the object a name for accessing its resources and adjust its shape using the resize box.

To load an object or widget from a palette as a new drawing, *Ctrl*-click on the widget's icon in the palette. The current drawing will be discarded (after a prompt) and the widget's drawing will be loaded. The run command will also be set to match the widget's resources. This is a convenient way to create a drawing containing just one widget. The widget's viewport is named *$Widget* by default, so the drawing may be saved and used with a GLG program. The drawing also contains a small icon viewport used by the Builder when the widget is shown in the palette. This icon will be ignored at run time, since only the *$Widget* viewport will be used.

Some palette items, such as viewportless dials, define a collection of graphical objects without a viewport. Such drawings do not define the *$Widget* viewport and can not be displayed by themselves, they need to be inserted into a viewport to be displayed.

When no optional palettes are installed, the *Palette* Menu provides the *Custom Objects*, *HMI Editor Widget Samples, Read Palette* and *Read Directory* options.

Custom Objects

The *Custom Objects* palette displays samples of pre-built objects that can be used in a drawing. It contains buttons, sliders and other objects, described in more detail in the *GLG Builder and Animation Tutorial*.

You can add your own objects to the Custom Object Palette. To do so, simply save it in the *widgets/custom_objects* directory. The object will be automatically added to the *Custom Objects* Palette when the builder is restarted or the palette is re-opened.

HMI Editor Widget Samples

The *HMI Editor Widget Samples* palette displays samples of HMI components that use application-specific public properties. These components are used with HMI Configurator, where public properties are used for simplified editing of a component.

Click on a component in the palette to add it to the drawing, then use *Object, Public Properties* to display its public properties.

Read Palette

The *Read Palette* option can be used to load a palette into the Builder. A palette is defined by a *Palette Description File* with the *.pal* extension. This file provides information about the palette and the objects it contains. Refer to the *Palette Description File Format* section and the *Adding Custom Palettes to the Builder* section for more information.

To load a palette, select *Read Palette*, then select the palette's *.pal* file using the activated file browser.

Read Directory

The *Read Directory* option can be used to scan a directory containing GLG drawings and display a palette containing all drawings in the directory. To read a directory, select *Read Directory,* then select a directory to read with the activated file browser.

Adding Custom Widgets and Custom Palettes

Naming Conventions for Palette Drawings

Each drawing file defines one palette item. A drawing may contain special graphics to be used as a palette icon, as well as the graphical object to be used in the Builder. The following describes the naming conventions used to annotate the icon and the graphical object to be used.

$Icon

By default, the complete content of the each drawing is added to the palette, which may take a considerable amount of memory and CPU time to load and render.

To make the process faster, each drawing can contain a *$Icon* resource, optimized in size and appearance for being displayed as a small palette item. If this resource exists in the drawing, it will be displayed as a palette icon. The icon will be used as is, without scaling, and must have a proper size and position to be displayed properly. The icons of the drawings supplied with the Toolkit define icons in a separate small viewport, so that they appear in the drawing the same way they will look like in the palette.

Alternatively, a D custom property named *$GlgScaleIconToFit* may be added to the icon and set to 1 to enable icon fitting, in which case the icon will be automatically scaled and positioned inside its palette slot. Icon scaling may not be very precise for icons that contain text objects, since the text objects do not scale well.

If the *$Icon* resource is absent, the whole drawing will be used as an icon. The drawing will be scaled to fit the icon area in the palette.

$Drawing and $Widget

The *$Drawing* or *$Widget* resource may be defined in the drawing to annotate the object or part of the drawing to insert when the palette icon is selected. If these resources are absent, the content of the whole drawing will be inserted.

The *$Widget* resource name is used for components that are contained in a viewport and may be used as a widget in a stand-alone way. *$Drawing* is used for components without a viewport. Such components must be placed in a viewport in order to be displayed.

Palette Description File Format

The Palette Description File (*.pal* extension) provides information about the palette and the objects it contains. Each line of the palette description file contains a key word and a value, separated with the *"="* sign. The following keywords are supported:

title
Specifies a mandatory title for the palette, which is displayed in the Builder's *Palettes* Menu.

num columns
Defines an optional number of columns displayed in the palette. The default value is 4.

num rows
Defines an optional number of visible rows displayed in the palette. By default, the palettes height is extended to accommodate as many rows as required to display all palette objects. If this parameter is set to a smaller value, for example 4, the palette will show only 4 rows and will include a scroll bar to scroll the rest of the palette objects.

background color
Specifies the palette's optional background color. This color is defined by supplying an RGB value in the range from 0 to 1. For example, use *"background color=1. 1. 1."* to define a white background. The default color is the color of the template.

directory
An optional directory parameter. If defined, all drawing files from the directory are displayed in the palette. Each drawing file defines one palette item and may contain the $Icon, $Drawing or $Widget resources (as described in the *Read Directory* section) to specify the icon to display and the object to insert in the drawing when the palette icon is selected. If the *directory* is not defined, the *entries* parameter is used. The Builder's *Custom Objects* Palette uses the *directory* parameter to define its entries.

entries
Specifies the drawing files to display in the palette. The entries parameter should be the last in the file and should contain nothing on the right side of the "=" sign. The drawing files are listed one per line on the lines following the *entries* key word.

Each drawing file defines one palette item and may contain the $Icon, $Drawing or $Widget resources (as described in the *Read Directory* section) to specify the icon to display and the object to insert in the drawing when the palette icon is selected. File names may include directory path relative to the location of the palette description file.

The palette file may also contain comments (lines starting with the *"#"* character).

Adding Custom Palettes to the Builder

The Builder uses the *palettes.pls* file located in GLG's *widgets* directory to detect installed palettes during start up. This file contains a list of palettes to be added to the Builder's *Palettes* Menu. To add a new custom palette to the Builder, create its palette description file and add it to the *palettes.pls* file. After restarting the Builder, the new palette will show up in the Builder's palette list.

Each line of the *palettes.pls* file contains the file name of a palette description file, including a path name relative to the location of *palettes.pls*. The file may also contain separator lines (lines that have just the *"-"* symbol) as well as comments (lines starting with the *"#"* character).

By default, on Unix, the Builder searches for the *palettes.pls* file in the GLG's *"widgets"* directory by using the *"../widgets"* path relative to the Builder's directory, or relatively to the current directory. On both Unix and Windows, the value of the GLG_DIR environment variable, if it is set, is used as a name of the directory that contains the "*widgets*" directory. You can change the default place where the Builder searches for the *palettes.pls* file by setting the GLG_PALETTES_LOCATION environment variable to point to either a new palette file name or a new directory where the *palettes.pls* file is located.

Edit

The *Edit* Menu provides options that let you make and manipulate copies of objects.

These options operate on the selected object. To operate on several objects at the same time, use the menu options for selecting multiple objects, then use editing options.

The *Cut*, *Copy*, and *Paste* options add and remove objects. The *Clone* options let you position and transform the added objects as they are created.

Undo

Undo reverts the effect of the last editing operation, such as changing geometry or an attribute value of an object or group of objects, changes to object's control points, layout and alignment operations, zooming and panning and others. The undo button displays the last editing operation that can be undone and changes its label to *Redo* after performing the *Undo* operation. Some advanced editing operations, such as exploding, constraining and some others, cannot be undone.

Undo History

Undo History displays a list of recent editing actions and allows selecting individual actions to undo or redo. Some geometry editing actions require the drawing's viewing state (pan and zoom) to be unchanged in order to be reverted properly. The changes to the drawing's viewing state are listed in the *Undo History* list and can be undone as well.

Select Multiple Objects

Select Multiple Objects is equivalent to using *Ctrl*-click. It starts multiple object selection without the need to hold the *Ctrl* key. After the option is selected, click on the objects in the drawing to add or delete them from the selection.

Select Rectangular Area

Select Rectangular Area is equivalent to clicking and dragging the mouse in the drawing to define the selection rectangle. It provides a convenient option for starting a rectangular selection for cases when all drawing area is covered with objects and there is no free space to click and drag the mouse without selecting some object. After the option is selected, click and drag the mouse anywhere in the drawing area to define the selection rectangle. All objects that are either completely or partially enclosed by the rectangle will be selected.

Select Object Inside Group

Select Object Inside Group is used to select an object inside a permanent group for editing in-place. To select an object inside the selected permanent group, select this menu option, then click on an object in the group. This activates what is called "group zooming". To select other objects in the group with the group zooming active, simply click on them with the mouse. To abort group zooming, press *Escape* or click on an empty area in the drawing. The *Ctrl*-*Shift*-click shortcut may also be used as a faster alternative. Refer to the the *Select Next section* on page 315 for more information.

Select All

Select All selects all objects in the drawing.

Cut

Cut removes the selected object from the drawing and places it on the clipboard. The cut object remains on the clipboard until you replace it by performing another *Edit*, *Cut* or an *Edit*, *Copy* operation.

You can retrieve the cut object by using *Edit*, *Paste*.

Copy

Copy places a *full* copy of the selected object on the clipboard without removing it from the drawing. The copied object remains on the clipboard until you replace it by performing an *Edit*, *Cut* or *Edit*, *Copy* operation.

You can retrieve the copied object by using *Edit*, *Paste*.

Paste

Paste gets a cut or copied object from the clipboard and adds it to the current drawing. Pasting an object does not delete the object from the clipboard, so you can paste the same object repeatedly. If an object is cut and then pasted repeatedly, the first paste places the object itself, preserving all constraints. Any consequent paste operations place a *full* copy of the object, removing any constraints.

The pasted object is added at the position of the current editing focus. For example, if an editing focus is inside of a viewport, the new object appears inside the viewport. If it's inside a group, the new object becomes a member of the group.

Delete

Delete removes the selected object from the drawing. The deleted object is irrevocably discarded; it is *not* placed on the clipboard.

To remove an object from the drawing and place it on the clipboard, use *Edit*, *Cut*. This allows you to move an object from one part of the drawing to another.

Define Clone Offset

Define Clone Offset determines a linear offset used for the placement of a clone with respect to the original object.

When you select *Define Clone Offset*, the Builder prompts you to click on two points that define the clone path. The clone path specifies the distance and direction from the origin of the original object to the origin of the clone, using the object's coordinate system.

The Builder uses a default offset of 50 units to the Southeast for all objects unless you redefine the offset.

To clone an object, use the clone options on the Edit menu (*Full Clone*, *Strong Clone, Weak Clone* and *Constrained Clone*).

Define Clone Transformation

Define Clone Transformation specifies a transformation to be used for positioning a cloned object. The original object is copied *and transformed* to produce the clone.

By default, the Builder applies the default linear clone offset. The *Define Clone Transformation* option may be used to define any transformation to offset the copies, for example rotate or scale.

To create the clones, use the clone options on the Edit menu (*Full Clone*, *Strong Clone, Weak Clone* and *Constrained Clone*).

To disable the clone transformation, select any object and use *Define Clone Transformation*, setting the parameters of the transformation so that they have a neutral effect. For example, for a move transformation you would set the move distance parameters to zero.

The clone transformation is applied to copies using *Transform Points,* changing the coordinates of their control points irrevocably, without attaching a transformation object to the clone. However, some objects such as circles, arcs and reference objects are treated differently for *Scale* and *Rotate* transformations. Cloning a circle using a rotate transformation attaches a static matrix transformation to the cloned object to position it.

When *Constrained Clone* is used, the control points of the clone are constrained to the points of the original object, and *Transform Points* would transform both the clone and the original. To avoid that, *Constrained Clone* attaches a static matrix transformation to the cloned object to position it and sets the clone's *MoveMode* to *MoveByXform*.

Full Clone

Full Clone creates a copy of the selected object. The copy has all the characteristics of the original object, including transformations, attributes, resources and internal constraints between its attributes. The *Full Clone* removes any attribute constraints to external objects.

The copy is created in the position specified by the current clone offset setting. If you specified a transformation using *Edit*, *Define Clone Transformation*, it is applied before the clone is drawn.

Weak Clone

Weak Clone creates a copy of the selected object preserving any internal constraints between the object's attributes. The *Weak Clone* also handles global attributes. The attributes of the object whose *Global* flag is set to *GLOBAL* are considered to be global attributes, and the corresponding attributes of the copy are constrained to the attributes of the original object.

The copy is created in the position specified by the current clone offset setting. If you specified a transformation using *Edit*, *Define Clone Transformation*, it is applied before the clone is drawn.

Strong Clone

Strong Clone creates a copy of the selected object. It also handles global attributes, but, unlike the weak clone, any attribute whose *Global* attribute is set to either *GLOBAL* or *SEMI-GLOBAL* is considered to be a global, and the corresponding attributes of the clone are constrained to the attributes of the original object.

The copy is created in the position specified by the current clone offset setting. If you specified a transformation using *Edit*, *Define Clone Transformation*, it is applied before the clone is drawn.

Constrained Clone

Constrained Clone creates a copy of the selected object. All the copy's attributes are constrained to the original object, so it has the same characteristics of the original, including transformations, attributes, and resources.

The copy is created in the position specified by the current clone offset setting. If you specified a transformation using *Edit*, *Define Clone Transformation*, it is applied before the clone is drawn. Because of the constraint, the offset and transformation are attached to the copy as static transformations.

When *Constrained Clone* is used, the control points of the clone are constrained to the points of the original object, and *Transform Points* would transform both the clone and the original. To avoid that, *Constrained Clone* attaches a static matrix transformation to the cloned object to position it and sets the clone's *MoveMode* to *MoveByXform*.

When you create a constrained clone, all the clone's attributes are constrained to the original object, regardless of the *Global* attribute settings. See page 258 for more information on the *Global* attribute.

Reset Scaling Xform

For a reference object, *Reset Scaling Xform* resets the internal transformation the reference object uses to stretch its instance, when the bounding box is stretched with the mouse. Selecting *Reset Scaling Xform* restores the original, unstretched appearance of the instance:

Add or Use Marked Object

Provides options for reusing rendering and text Box attributes, chart label and plot line attributes, as well as viewport's font tables and other objects. The menu becomes active when the objects to be reused are marked by selecting the *Mark* button in the *Object Properties* dialog. The *Attribute Clone Type* option of the *Options* menu controls the constrain type of the added copies. When a group is selected, adding a marked object adds copies of it to all objects in the group.

Rendering Attributes

Adds a copy of a marked rendering attributes object.

Box Attributes

Adds a copy of a marked text box attributes object.

Font Table

Adds a copy of a marked font table.

Light Attributes

Adds a copy of a light attributes object.

Background Attributes

Replaces chart background attributes with a copy of the marked attributes.

Grid Attributes

Replaces chart grid attributes with a copy of the marked attributes.

Cross-Hair Attributes

Replaces chart cross-hair cursor attributes with a copy of the marked attributes.

Tick Attributes

Replaces axis tick attributes with a copy of the marked attributes.

Line Attributes

Replaces attributes of a chart's plot or level line with a copy of the marked attributes.

Tick/Legend Label Attributes

Replaces axis or legend tick attributes with a copy of the marked attributes.

Axis Label Attributes

Replaces axis label attributes with a copy of the marked attributes.

View

The View Menu provides options to let you change the appearance of the drawing. These options have no effect on the underlying drawing, but just alter its appearance within the drawing area.

Set View

The Set View submenu provides options to let you change the projection used to display the drawing.

Main

In this view, the X axis points to the right, the Y axis points up, and the Z axis points toward the viewer.

Back

In this view, the X axis points to the left, the Y axis points up, and the Z axis points away from the viewer.

Left

In this view, the X axis points away from the viewer, the Y axis points up, and the Z axis points to the right.

Right

In this view, the X axis points toward the viewer, the Y axis points up, and the Z axis points to the left.

Top

In this view, the X axis points to the right, the Y axis points toward the viewer, and the Z axis points down:

Bottom

In this view, the X axis points to the right, the Y axis points away from the viewer, and the Z axis points up.

Adjust View

Adjust View defines and applies a transformation to the view of the drawing, letting you transform the view incrementally.

The view transformations are defined and applied in the same way as object and clone transformations; see the *GLG Objects* chapter for transformation descriptions. A viewing transformation does not affect the drawing's content; it only adjusts the viewing projection used to present the drawing.

To return to one of the predefined projections, select a view from the *View*, *Set* submenu.

When the view is adjusted in the Builder, the result of the adjustment is stored in the viewport's *Zoom* transformation. If you want to transform the view from a program, attach a parametric view transformation to the viewport object. See the *Viewport* section of the *GLG Objects* chapter for more information on *Zoom* and *View* transformations.

Load View Transformation

Load View Transformation applies a saved view transformation to the current drawing. The transformation does not affect the drawing itself, just the way it is presented.

When you select *Load View Transformation*, the Builder prompts you to enter a file name, using the standard file selection dialog.

To save a view projection, use *View*, *Save View Transformation*.

Save View Transformation

Save View Transformation writes the definition for the current view projection to a file. A saved view transformation is useful if you frequently use a particular projection that is not among the standard views. You can configure your projection once, save it, and then load it whenever you want to view a drawing using that projection.

When you select *Save View Transformation*, the Builder prompts you to enter a file name, using the standard file selection dialog.

To apply a saved view projection, use *View*, *Load View Transformation*.

Coordinate System

The *Coordinate System* submenu provides options to let you view the drawing using different coordinate systems. The effect of these options depends on the relationships among the objects in the drawing. If the objects share the same coordinate system, the options have no effect on the appearance of the drawing.

Changing the viewing coordinate system does not affect the objects or their relationships to one another. It affects only the Rotation and Scaling axes and the way the Builder interprets numerical values of coordinates. For information about the various coordinate systems relevant to a GLG drawing, please see the *Structure of a GLG Drawing* chapter.

View

View lets you edit the drawing using the coordinate system of the viewport, before view transformations are applied.

Drawing

Drawing lets you edit the drawing using the coordinate system for the drawing, after view transformations are applied.

Parent

Parent lets you edit the drawing using the coordinate system of the selected object's parent.

Object

Object lets you edit the drawing using the selected object's coordinate system.

Zooming

The Zooming submenu provides options to let you change the scale of the view. The changes in scale affect your view of the drawing, not the drawing itself, and are saved with the viewport. All the zooming options use the drawing's coordinate system. The Builder's *Control Panel* also provides zooming controls.

Zoom In

Zoom In increases the scale of the drawing, so the objects look bigger. Zooming into a drawing enlarges the objects in the center of the drawing, but objects near the edges of the drawing may be clipped.

To control the degree of scaling, use *View*, *Zooming*, *Set Zoom Factor*.

Zoom Out

Zoom Out decreases the scale of the drawing, so the objects look smaller.

To control the degree of scaling, use *View*, *Zooming*, *Set Zoom Factor*.

Zoom To

Zoom To zooms into a specified area of the drawing. When you select this option, the Builder prompts you to specify two points that define a bounding box for the zoom area.

To recover your view of the excluded area, use *View*, *Zooming*, *Zoom Out*.

Set Zoom Factor

Set Zoom Factor controls the scale factor to be applied when you zoom using the *Zoom In* and *Zoom Out* buttons of the *Control Panel* or the menu options. The Builder prompts you for a scaling factor (for example, 2 to scale up or 0.5 to scale down).

Reset Zoom

Reset Zoom returns to the normal scale for the view (100% zoom), and resets any changes in the projection.

Pan To

Pan To moves the center of the view to another part of the drawing area.

After you select this option, click on a spot to use as the new center of the view.

Scroll by Dragging

Starts dragging mode. Click and drag the mouse after selecting this option to scroll the drawing with the mouse.

Scrolling the drawing with the mouse may also be performed by the *Ctrl-click-drag* sequence in any empty area of the drawing. However, if the drawing area is completely occupied by objects, this menu option provides an alternative.

Traverse

The *Traverse* Menu provides options to let you work with the object hierarchy.

Hierarchy Down

Hierarchy Down shows the members of the object hierarchy below the selected object. The effect of this option depends on the selected object:

- For a container object such as a group or viewport, this option shows the elements inside the container object.
- For a composite object such as a series or object reference, this option shows the template for the composite object.
- For the file reference (subdrawing), this option loads the referenced file.
- For polylines and polysurfaces this option shows the template marker object.

For non-composite objects, this option is grayed.

This menu option is equivalent to the *Hierarchy Down* button at the left side of the Builder window.

To navigate back up through the object hierarchy, use *Traverse*, *Up*.

Transformation Down

For a transformed object, *Transformation Down* shows the original object before its transformation.

To return to a view of the transformed object, use *Traverse*, *Up*.

Up

Up undoes the effect of *Traverse*, *Hierarchy Down*, returning to a higher level of the object hierarchy. It also undoes the effect of *Traverse*, *Transformation Down*, returning to a view of the transformed object.

Set Focus

Set Focus enters a mode that makes a viewport's contents available for editing without filling the whole Builder window. This lets you edit the contents of a viewport in the context of the surrounding drawing, unlike *Hierarchy Down*, which excludes from the drawing area any objects not within the selected branch of the object hierarchy. When the focus is set to a viewport different from the main drawing area, *Traversing Down* is disabled until the focus is returned to the main drawing.

To set the focus, use this menu option (or click on the *Set Focus* button on the *Control Panel*). The Builder prompts you to select a viewport to focus on. *Ctrl*-*Shift*-clicking on a viewport also moves focus into it, acting as a convenient shortcut for *Set Focus*.

To return to the default editing mode, use *Traverse*, *Main Focus*, or click on the *Main Focus* button.

Main Focus

Main Focus returns focus to the main drawing area. *Ctrl-Shift*-clicking on a top-level viewport of the drawing area also moves focus into it.

Main Focus terminates the Set Focus mode. To edit the contents of the selected viewport, use *Traverse*, *Hierarchy Down*.

This option is grayed until you use *Traverse*, *Set Focus* to change the viewport editing mode.

Select Next

When you select a member of a group with a mouse click, you select the entire group. If you want to edit only one member of the group, you can use *Select Next* to select it. This option selects members of a group that exist at the next lower level of the object hierarchy. That is, if one of the members of the group is itself a group, when you choose *Select Next* and then click on a member of that sub-group, you select the entire sub-group. If you are editing nested groups, you can use *Select Bottom* to select objects at the bottom of the hierarchy of nested groups. When a permanent group is selected, the *Ctrl-Shift*-click on an object in the group selects the object, acting as a shortcut for *Select Next*. When an object inside the group is selected using *Select Next*, the boundaries of its parent group are highlighted with a dotted line to provide visual feedback for the traversal of the group hierarchy.

To change attributes that are common to all the members of a group, use *Traverse*, *Edit All*, or the *Edit All* button on the group's *Properties* dialog.

To delete a group object and separate its members for independent editing, select the group and use *Arrange*, *Explode, Object*.

Alternatively, you can select the group, and use *Traverse*, *Hierarchy Down* to move to the hierarchy level that shows the individual objects in the group. At this level, you can select and edit each member of the group independently. To return to the hierarchy level that shows the group object, use *Traverse*, *Up*.

This option is equivalent to the *Select Next* button in the *Selected Object Properties* dialog for a group object.

Select Next mode is aborted when an object outside of the group is selected.

Select Bottom

When you select a member of a group with a mouse click, you select the entire group. If you want to edit only one member of the group, you can use *Select Bottom* to select it. This option selects members of a group that exist at the lowest visible level of the object hierarchy. That is, if one of the members of the group is itself a group, when you choose *Select Bottom* and then click on a member of that sub-group, you select only the object on which you clicked. If you want to select the entire sub-group instead, you can use *Select Next* to select objects at the next lower level of the

hierarchy of nested groups. When an object inside the group is selected using *Select Bottom*, the boundaries of its parent group are highlighted with a dotted line to provide visual feedback for the traversal of the group hierarchy.

To change attributes that are common to all the members of a group, use *Traverse*, *Edit All*, or the *Edit All* button on the group's *Properties* dialog.

To delete a group object and separate its members for independent editing, select the group and use *Arrange*, *Explode, Object*.

Alternatively, you can select the group, and use *Traverse*, *Hierarchy Down* repeatedly to move to the lowest level of the hierarchy. At this level, you can select and edit each member of the group independently. To return to the hierarchy level that shows the group object, use *Traverse*, *Up*.

This option is equivalent to the *Select Bottom* button in the *Selected Object Properties* dialog for a group object.

Select Bottom mode is aborted when an object outside of the group is selected.

Edit All (First)

For a group object, *Edit All (First)* starts editing the attributes of objects in the group by using the first object in the group to select a set of attributes for editing. This is a convenient option for fast editing of groups that contain objects of the same type.

For groups that contain objects of different types, the *Edit All (Select)* option allows you to select a set of attributes to edit. For example, if the group contains both the polygon and text objects, the *Edit All (Select)* option allows you to select the polygon or text attributes to be edited.

If a *Font Table*, *Color Table*, *Rendering Attributes* or *Box Attributes* are added to an object in a group in the *Edit All* mode, constrained copies are added. Changing any attribute will affect all copies. Individual or all attributes of added copies may be unconstrained. Use the *Unconstrain* button in the *Attribute* dialog to unconstrain selected attributes of the constrained objects (in *Edit All* mode, the attribute will be unconstrained from all copies).

Edit All (Select)

For a group object, *Edit All (Select)* allows you to choose a set of attributes to edit. This may be convenient when the group contains objects of different types with different sets of attributes.

When you select this option, the Builder prompts you to select an object that has the attributes you want to change. The changes you make to an attribute in the *Properties* dialog apply to all the objects in the group that have the attribute.

For example, consider a group that contains two circles, a parallelogram, and a fixed text object. Select *Edit All (Select)* and then select a circle. The *Properties* dialog shows the attributes for a circle object. Resetting the *Resolution* affects both circles. Resetting the *LineWidth* affects both of the circles and the parallelogram. However, the text object doesn't have a *LineWidth* attribute and is not affected.

If a *Font Table*, *Color Table*, *Rendering Attributes* or *Box Attributes* are added to an object in a group in the *Edit All* mode, constrained copies are added. Changing any attribute will affect all copies. Individual or all attributes of added copies may be unconstrained. Use the *Unconstrain* button in the *Attribute* dialog to unconstrain selected attributes of the constrained objects (in *Edit All* mode, the attribute will be unconstrained from all copies).

Arrange

The *Arrange* Menu provides options to let you change the relationships between objects.

Create Permanent Group

Create Permanent Group creates a group object, which is a container for objects. A group is an object that organizes the objects it contains, letting you apply actions to all the objects at once. A group can contain any object, including other groups.

When you use this option, the Builder prompts you to click and drag the mouse in the drawing to define a rectangle that touches or encloses all the objects to be included in the group. A group object does not appear as a visible shape, but the control points of objects in a group appear as hollow squares. Clicking on any member of a group selects the group. This option is equivalent to the *Group* icon on the *Object Palette*.

To remove an object from a group, use the options on the *Arrange*, *Explode* submenu.

To edit a single object in a group, use the options on the *Traverse* menu (*Edit Next*, *Edit Bottom*, and *Edit All*; see page 315). Alternatively, use *Traverse*, *Hierarchy Down* to edit the members of the group; see page 314. The *Ctrl-Shift-Click* on an object in a group may be used as a shortcut for getting access to objects inside the group.

Create Temporary Group

Create Temporary Group is equivalent to *Select Rectangular Area* described on page 306.

Select Multiple Objects

Select Multiple Objects is equivalent to *Select Multiple Objects* described on page 306.

Add Object to Group

Add Object to Group adds an object to the selected group object. It may be used with both temporary and permanent groups.

Use the following procedure:

1. Select the group.
2. Select *Arrange*, *Add Object to Group*.
3. Click on the first object to add to the group.
4. Click on the next object to add to the group.
5. When you have added all the objects to the group, use the *Esc* key or the right mouse button to complete the operation.

The status bar at the bottom of the Builder window provides prompts to guide you through the procedure.

This option is equivalent to the *Add Object* button in the *Selected Object Properties* dialog for a group object.

Delete Object from Group

Delete Object from Group removes a single object from the selected group. The option may be used with both temporary and permanent groups.

Use the following procedure:

1. Select the group.
2. Select *Arrange*, *Delete Object from Group*.
3. Click on the first object to remove from the group.
4. Click on the next object to remove from the group.
5. When you have removed the objects from the group, use the *Esc* key or the right mouse button to complete the operation.

The status bar at the bottom of the Builder window provides prompts to guide you through the procedure.

This option is equivalent to the *Remove Object* button in the *Selected Object Properties* dialog for a group object.

Add or Delete Object from Group

Add or Delete Object from Group adds an object to the group if the object is not part of the group, or deletes the object from the group if the object is a part of the group.The option may be used with both temporary and permanent groups.

Use the following procedure:

1. Select the group.
2. Select *Arrange*, *Add or Delete Object from Group*.
3. Click on the first object to add or delete from the group.
4. Click on the next object to add or delete from the group.
5. When you have removed the objects from the group, use the *Esc* key or the right mouse button to complete the operation.

The status bar at the bottom of the Builder window provides prompts to guide you through the procedure.

For temporary groups, the same action can be accomplished by *Ctrl*-clicking on the object with the mouse.

Select Next

Select Next is equivalent to *Select Next* on page 315.

Select Bottom

Select Bottom is equivalent to *Select Bottom* on page 315.

Edit All (First)

Same as the corresponding option of the *Traverse* menu.

EditAll (Select)

Same as the corresponding option of the *Traverse* menu.

Permanent Group

Permanent Group toggles the type of the selected group object between temporary and permanent. The current group type is displayed as the state of this toggle button.

Explode

The *Explode* submenu provides options to let you separate the objects in a composite object such as a group or series, so that they become independent objects.

The effect of *Explode* depends on what kind of object is selected. In general, it simplifies the object's representation by removing one level of the object hierarchy:

- For a series, *Explode* deletes the series object and replaces it with a group with same name as the exploded series object; the group will contain members of the series.
- For a reference, *Explode* deletes the reference object and replaces it with an instance of the template object. For file references (subdrawings), the loaded instance of a drawing is used.
- For a group, *Explode* deletes the group object. The objects that were previously members of the group become independent objects.
- For a connector object, *Explode* replaces the connector with the object used to render connector's graphics: a polygon for recta-linear connectors, or an arc for arc connectors.
- For a circle, a parallelogram, a rectangle, a rounded rectangle or a spline, *Explode* replaces the exploded object with a polygon that has a control point at each vertex. For example, exploding a circle creates a many-sided polygon, and exploding a rectangle creates a four-sided, unconstrained polygon.

When an object other than a group is exploded, any actions and custom properties attached to the object are transferred to the object used as a replacement.

Object

Object explodes the selected object. If the selected object is a group or series, this option only affects the top level of the association. Use *Arrange*, *Explode*, *Sub-Objects* to explode the sub-objects.

If the selected object is a group with attached transformations, the transformations are copied to the resulting objects. The Builder prompts you for the type of cloning used to copy the transformations.

Sub-Objects

For objects that are contained in a group, *Sub-Objects* lets you explode the sub-objects without affecting the group.

Xform

When you select this option, the Builder prompts you to choose between transforming the points in each object or adding the transformation to each object's control points. If you choose to add the transformation to the control points, the Builder prompts you for the type of cloning used to copy the transformations.

This option is grayed if the selected group object has no transformations.

Sub-Object Xforms

Sub-Object Xforms parallels the function of *Xform*, affecting the sub-objects in the group.

Reorder

The *Reorder* submenu provides options to let you change the drawing order of objects. Unless the parent viewport or group has the *DepthSort* attribute turned on, the last object drawn appears to be in front of any other overlapping objects. This is true regardless of the spatial positions of the objects, with the exception of the viewports which are windows and always appear on top of graphical objects.

Move to Back

Move to Back moves the selected object behind other objects in the drawing. All overlapping objects appear to be in front of the selected object.

Bring to Front

Bring to Front moves the selected object in front of other objects in the drawing. All overlapping objects appear to be behind the selected object.

Move Backward

Move Backward changes the selected object's place in the rendering list, moving the object back in the list by one position every time Move Backward is selected (the *Ctrl*-+ accelerator may also be used).

Move Forward

Move Forward changes the selected object's place in the rendering list, moving the object forward in the list by one position every time Move Forward is selected (the *Ctrl*-- accelerator may also be used).

Replace Viewport with SubWindow

Replaces the currently selected viewport with a *SubWindow* object that uses the same control points as the original viewport. The operation preserves any constraints on the control points. The *Source* attribute of the subwindow is set to INCLUDED and the original viewport is assigned as the subwindow's template. To use the subwindow for displaying viewports form drawing files, change its *Source* to FILE and set the *Source Path* to the filename of a drawing to be displayed in the subwindow.

Polygon Points

Contains options for reordering or adding points to a polygon.

Inverse Polygon Points

Inverts the order of the polygon's control points. This does not change the polygon's appearance, but the first point in the list of the polygon's points (annotated in the drawing with the small black square) becomes the last point in the list. The list of the polygon points may be edited by pressing the *Point List* button in polygon's *Properties* dialog. Inverting the points' order may be convenient when merging polygons.

Add Polygon Points

This submenu provides options for merging polygons. To merge polygons, all points of one polygon are first marked and then added to another polygon. To mark polygon's points for merging, select the polygon, display its *Properties* dialog, press the *Point List* button to display its point list and press *Mark List*.

Add To Beginning

Adds the marked list of points at the beginning of the point list of the selected polygon.

Add To Beginning Reversed

Adds the marked list of points at the beginning of the point list of the selected polygon, reversing the order of points of the marked point list so that the first point is added last.

Add To End

Adds the marked list of points at the end of the point list of the selected polygon.

Add To End Reversed

Adds the marked list of points at the end of the point list of the selected polygon, reversing the order of points of the marked point list so that the first point is added last.

Template

Provides options for managing templates of subdrawings, subwindows and series objects.

Mark Template

Stores a template of the selected *SubDrawing* or *SubWindow* object to be reused. The option is enabled only for *SubDrawing* and *SubWindow* objects that use included templates.

Use Marked Template

Replaces a template of the selected *SubDrawing* or *SubWindow* object with the template marked with the *Mark Template* option. The template will be shared between all subdrawings or subwindows that use this option. The option is enabled only for *SubDrawing* and *SubWindow* objects that use included templates.

Replace Parent's Template

For a series or reference object, *Replace Parent's Template* lets you use a different object as a template. Use the following procedure:

1. Select *Traverse*, *Hierarchy Down* to open the series object.
2. Select or create a new template object.
3. Select *Traverse*, *Replace Parent's Template*.

When you go back up the hierarchy, the series or reference is drawn with the new template object. The old template is discarded.

If the template is a simple object (a polygon, for example), this option may be used to add more objects to the template by replacing the polygon with the group containing other objects as well.

Legend

Contains options for managing chart legends.

Mark Legend

Marks the selected legend object.

Set Chart Legend

Sets a previously marked legend object as a legend of the selected chart.

Reset Chart Legend

Resets a legend of the selected chart object.

GIS Zoom Mode

Provides options for setting the zoom mode of a viewport to the GIS Zoom Mode.

Set as Parent Viewport's GIS Object

Sets the GIS Zoom Mode of the GIS Object's parent viewport by setting the currently selected GIS Object as the parent viewport's GIS Object. In the GIS Zoom Mode, the zoom and pan controls of the viewport zoom and pan the map displayed in the GIS Object instead of zooming and panning the viewport's drawing. The GIS Zoom Mode is persistent and is stored with the drawing. To unset the GIS Zoom Mode, use the *Unset GIS Zoom Mode* option described below.

Unset GIS Zoom Mode

Resets the GIS Zoom Mode of the selected viewport to the Drawing Zoom Mode.

Chart Zoom Mode

Provides options for setting the zoom mode of a viewport to the Chart Zoom Mode.

Set as Parent Viewport's Chart Object

Sets the Chart Zoom Mode of the chart's parent viewport by setting the currently selected chart object as the parent viewport's chart object. In the Chart Zoom Mode, the zoom and pan controls of the viewport zoom and scroll the data displayed in the chart object. The Chart Zoom Mode is persistent and is stored with the drawing. To unset the Chart Zoom Mode, use the *Unset Chart Zoom Mode* option described below.

Unset Chart Zoom Mode

Resets the Chart Zoom Mode of the selected viewport to the Drawing Zoom Mode.

Layout

The *Layout* Menu provides options to align and layout objects in the drawing. It may also be used to view or set the objects' width and height using the *Layout Toolbox* option.

Layout Toolbox

This option activates the *Layout Toolbox* which contains icons and controls for performing various align and layout operations. See the *Object Layout and Alignment* chapter on page 234 for more information.

Align

The *Align* submenu provides options for aligning objects.

Align Left

Align Left aligns the left edge of each selected object with the left edge of the anchor object. If no anchor object is selected, the left most selected object is used as an anchor.

Align Horiz. Center

Align Horiz. Center aligns the center of each selected object with the center of the anchor object horizontally. If no anchor object is selected, the left most selected object is used as an anchor.

Align Right

Align Right aligns the right edge of each selected object with the right edge of the anchor object. If no anchor object is selected, the right most selected object is used as an anchor.

Align Top

Align Top aligns the top edge of each selected object with the top edge of the anchor object. If no anchor object is selected, the highest selected object is used as an anchor.

Align Vert. Center

Align Vert. Center aligns the center of each selected objects with the center of the anchor object vertically. If no anchor object is selected, the highest selected object is used as an anchor.

Align Bottom

Align Bottom aligns the bottom of each selected object with the bottom of the anchor object. If no anchor object is selected, the lowest selected object is used as an anchor.

Make Same Size

Make Same Size submenu provides options for setting size of selected objects to be the same.

Width

Sets the width of each selected object to be the same as the width of the anchor object. If no anchor object is selected, the left most selected object is used as an anchor.

Height

Sets the height of each selected object to be the same as the height of the anchor object. If no anchor object is selected, the left most selected object is used as an anchor.

Both

Sets both the width and height of each selected object to be the same as the width and height of the anchor object. If no anchor object is selected, the left most selected object is used as an anchor.

Distribute

Distribute submenu provides options for distributing objects in the selected group, leaving no spaces between the objects.

Across

Distributes objects horizontally leaving no extra spaces between the object's extents.

Down

Distributes objects vertically leaving no extra spaces between the object's extents.

Space Evenly

Space Evenly submenu provides options for evenly distributing spaces between objects in the selected group.

Across

Distributes objects horizontally with equal spaces between the object's extents.

Down

Distributes objects vertically with equal spaces between the object's extents.

Distribute Evenly

Distribute Evenly submenu provides options for evenly distributing objects in the selected group using objects' centers.

Across

Distributes objects horizontally with equal distance between the objects' centers.

Down

Distributes objects vertically with equal distance between the objects' centers.

Select Anchor

Select Anchor defines the anchor object. Select this option, then click on the object with the mouse to define it to be the anchor. The anchor selection will be preserved until the currently selected group is unselected.

Align Points

When this toggle is checked, the objects' control points are used to align objects. If the toggle is unchecked, the objects' extents will be used.

More

This option is equivalent to the *Layout Toolbox* option: it brings the Layout Toolbox for more alignment and layout options.

Object

The *Object* Menu provides options to let you create and manipulate objects within the drawing.

Create

The options under the *Create* option let you add new objects to the drawing.

Most of the options under the *Create* submenu correspond to the buttons in the *Object Palette* on the left side of the Builder window. You can create the object by choosing its type from this menu, or by clicking on the corresponding button in the palette.

If an object has an icon in the drawing primitives palette, the icon is shown next to the object description. If an icon is not displayed, the object may be created only by using the *Object*, *Create* main menu.

See page 225 for basic instructions on drawing objects.

Polygon

The *Polygon* options let you create the following kinds of polygons:

- Open polygon 

- Closed polygon 
- Filled polygon 

To create any type of polygon, click on each point to be used as a vertex, then click the right mouse button (or use the Esc key) to complete the polygon. For the closed and filled polygons, the Builder joins the first and last vertices, closing the polygon.

Rectangle

The *Rectangle* options let you create the following kinds of rectangles:

- Rectangle 
- Filled rectangle 

Although the GLG object set does not include a rectangle object, the *Rectangle* options are provided for convenience. Drawing a rectangle actually creates a specialized parallelogram with perpendicular sides. The sides are not constrained to remain perpendicular, though they remain parallel unless you explode the object.

To create a rectangle, click on two points to define the diagonal corners of the rectangle.

Rounded Rectangle

The *Rounded Rectangle* options let you create the following kinds of rectangles:

- Rounded Rectangle
- Filled rounded rectangle 

To create a rounded rectangle, click on two points to define the diagonal corners of the rectangle. It creates a parallelogram with perpendicular sides and rounded corners. The size of the rounded corners is controlled by the object's *Radius1* and *Radius2* attributes.

Parallelogram

The *Parallelogram* options let you create the following kinds of parallelograms:

- Parallelogram
- Filled parallelogram

To create a parallelogram, click on one point, and then click on two other points. The second and third points define two vectors from the first point; they specify the angles and lengths of the opposing sides. The opposing sides of the parallelogram remain parallel when the object's control points are moved. To remove this constraint, explode the object.

Arc

The *Arc* options let you create the following kinds of arcs:

- Arc 
- Chord arc 
- Segment arc 
- Filled chord arc 
- Filled segment arc 

An arc is a many-sided regular polygon, like a circle; however, an arc does not encompass 360°. A simple arc is just the segment of a circle's perimeter connecting the points you choose. A chord arc closes the shape with a straight line from one end of the arc to the other. A segment arc closes the shape with two straight lines from each end of the arc to the center, like a wedge of pie.

To create any type of arc, click on the center point, a point to define the radius, and a point to define the angle of the sector.

You can convert an arc to a circle by editing its *StartAngle* or *EndAngle* attributes.

The control points of an arc or a circle are unusual for graphical objects. See the description of a circle (below) for an explanation.

Circle

The *Circle* options let you create the following kinds of circles:

- Circle 
- Filled circle 

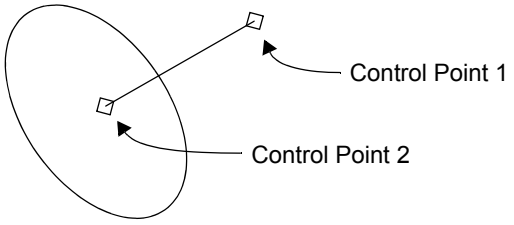
A circle is a many-sided regular polygon. Its *Resolution* attribute specifies the number of sides used to render the circle. At low *Resolution* values, the shape no longer resembles a circle; for example, a circle with a *Resolution* of 5 is a pentagon.

To define a circle, click on the center point, then on a point to define the radius.

In GLG, both circles and arcs are rendered using an *Arc* object type. A circle is a special case of an arc with a *StartAngle* of 0 and an *EndAngle* of 360.

Circles and arcs have an arrangement of control points different from other graphical objects. Each circle has two control points that initially appear superimposed at the center of the circle. (The move point is moved slightly away from the center to avoid confusion.) The two points define a line to

which the circle is perpendicular. The length of the line is ignored. This means that you can grab one of the control points, and tilt the circle by moving it. However, a circle has no control points on its perimeter for resizing it; to change its radius, change its *Radius* attribute or use the *Resize Box*.



The Control Points of a Circle

Ellipse

The *Ellipse* options let you create the following kinds of ellipses:

- Ellipse
- Filled ellipse 

An ellipse is a rounded rectangle with rounded corners taking the whole extent of the rectangle. Its *Resolution* attribute specifies the number of segments used to render each corner of the ellipse. At low *Resolution* values, the shape no longer resembles an ellipse; for example, an ellipse with a *Resolution* of 2 is a hexagon.

To create an ellipse, click on two points to define the diagonal corners of the ellipse's bounding rectangle.

Spline

The *Spline* options let you create a multi-point Bezier cubic spline used to render curves in 2D or 3D space:

- Spline 
- Filled spline 

To create a spline, click in the drawing area to define the number of spline control points, then click the middle or right mouse buttons (or use the Esc key) to complete the spline. The spline will render a smooth curve defined by the control points. The shape of the curve may be changed by moving the control points.

Marker

A marker is an object that indicates the position of a single point. Markers are made by selecting one or more shapes from a set of predefined shapes such as squares, crosses, and circles. Unlike other graphical objects, marker objects do not change their size when the viewport is resized. 

To create a marker, select *Object, Create, Marker* and then specify the point.

Since markers are always drawn the same size, they are not affected by transformations. To change a marker's size, change its *Size* attribute.

Image

An image object may be used to display an image in GIF, JPEG, PNG or BMP (on Windows only) formats. The *Image* options let you create the following kinds of images:

- Fixed Size Image 
- Scalable Image 

To create an image, select the type of the image: fixed size or scalable, define its position (one control point for the fixed image and two points for the scalable image), and select the image file.

For images of fixed size, the *Anchoring* attribute may be used to control the image's position relative to its control point.

Text

The *Text* options let you create the following kinds of text objects:

- Fixed Text presents a string. To create a fixed text object, click in the drawing to define the text's position, then type the string in the text entry box. 
- Scaled Text presents a string within a bounding box. Resizing the bounding box changes the font size of the string. To create a scaled text object, click on two points to define the diagonal corners of the bounding box, then type the string in the text entry box. Note that the text is scaled by selecting different size fonts from the viewport's font table, not by changing the dimensions of the characters. Depending on the selection of sizes defined in the font table, this may appear to give incorrect results when the text object is resized. 
- Spaced Text presents a string within a bounding box, with flexible orientation. Changing the box repositions the string. To create a spaced text object, click on two points to define a line, then click on a third point to define the height of the bounding box. Finally, type the string in the text entry box.  . The third control point is used for line positioning when the text object has several lines.

A text object is a graphical object that presents a string. The text object itself just sets the boundaries of the text. The text object's data attribute defines the string that appears in the drawing.

You can transform the text object itself using the transformations that apply to geometric data; the transformations affect the bounding box that contains the text. You can transform the text object's *String* attribute using any of the transformations that apply to string data; they can be used to display a numeric value or another string. For a complete list of transformations and details on text object types, see the *GLG Objects* chapter.

Font Availability

All the text objects can display any font that is available in the GLG Graphics Builder. The final appearance of the string depends on the font table attributes of the viewport that contains the text object. To set the font table attributes:

1. Select the viewport and use *Object*, *Properties* to see the viewport's attributes.

2. In the *Selected Object Properties* dialog for the *Viewport* object, click on the ellipsis button  for *More*, to see the attributes of the *Screen* object.

3. In the *Selected Object Properties* dialog for the *Screen* object, set the *Default Fonts* attribute to *NO*, and click on the ellipsis button  for *Fonts*, to see the attributes of the *Fonttable* object.

4. In the *Selected Object Properties* dialog for the *Fonttable* object, use the buttons to set the attributes for the *Fonttable* object. This object refers to a font table, which is an array of data that includes the font specification and available sizes.

5. To see the fonts in the selected font table, click on the ellipsis button  for the *Fonts* attribute and edit *Font* objects in the font table. Each font object allows defining fonts for both Windows, Unix and Java run-time environments, as well as for PostScript printing.

GIS Object

A GIS Object provides a way to embed GIS maps generated with the GLG Map Server into GLG drawings. It automatically handles all aspects of the low-level interaction with the Map Server to display, zoom and pan GIS map data. The GIS Object  provides attributes to control projection, center and extent of the map.

To create a GIS Object:

1. Select *Object*, *Create*, *GIS Object* and choose rectangular or orthographic projection to create the GIS Object.

2. Click on two points to define a rectangular area to use for the map display.

3. When prompted, select a dataset file that describes the GIS data to render.

The GIS Object will display the map specified by the dataset file. The map may be positioned by changing the *GISCenter* and *GISExtent* attributes of the object. The GIS Zoom mode of the drawing's Integrated Zoom feature may be used at run time to zoom and pan the map. Refer to the *Viewport* section of the *GLG Objects* chapter on page 84 for details.

Group

A group is a container for other objects. It lets you manipulate all members of the group as if they were a single object. 

To define a group, select *Object*, *Create*, *Group* and drag a rectangle through the objects you want to include in the group. Any object touched by the defining rectangle is included in the group. The groups created using this method are permanent. Refer to the *Multiple Selection* chapter on page 227 for information about temporary groups.

The *Arrange* and *Traverse* menus provide options to let you work with a group object. The *Arrange*, *Create Permanent Group* option is equivalent to the *Object*, *Create*, *Group* option.

You can use the Builder to create groups of graphical objects, but groups can contain any objects, graphical or not. The polygon, for example, contains a group of point coordinates. It is not uncommon to see non-graphical objects grouped using the GLG API.

Container

A container object is a wrapper around a group of objects; it encapsulates a collection of objects in a single entity and provides a single control point for positioning it in the drawing. A container's *Template* holds the objects drawn in the container. If the container is copied, the contained template object is copied as well, so that each copy of the container has its own independent template. The container draws its template directly, without creating any additional instances of it. 

Containers may be used to implement **node/edge** functionalities. If a container is used as a node, the container's single control point may be conveniently used for positioning or attaching connectors to. Containers may also be used to preserve center of rotational dynamics when objects are moved.

To create a container:

1. Select an object to use as a template.

2. Select *Object*, *Properties* to set the attributes of the template object so that its instance will inherit the appropriate settings. Name the object.

3. Click on the  button, or select *Object*, *Create*, *Container* menu option to place the selected object in a container, then click in the drawing to define the container's position.

When you create a container, the Builder places it at the current level of the hierarchy. The container's template appears in the resource hierarchy as the *Template* resource; it is also visible in the resource hierarchy under its original name.

When you copy a container, each copy will store its own copy of the template. If containers are used to represent connected nodes, a container's control point may be used for constraining connecting lines that represent edges.

To edit the container's template object, use *Traverse, Hierarchy Down*. When finished, use *Traverse, Up* to go back to the top level.

Viewport

A viewport is a rectangular object that acts as a container for other objects. The viewport represents the final stage in the visualization of a drawing object; it presents the objects within the drawing area and may be used as a widget in different run-time environments. 

To create a viewport object, click on two points to define the diagonal corners of the viewport. To place objects inside the viewport, use *Traverse*, *Hierarchy Down* to open the viewport, or use *View*, *Set Focus*; see page 314.

A drawing can contain multiple viewports, viewports nested within viewports, or no viewports at all. However, to use the drawing as a widget, the drawing must contain a viewport named *$Widget* that contains all the objects in the drawing; see the *Details of using GLG Standard API for C and C++* chapter.

Note: Due to features of the Windows graphical environment, there could be an inconsistency when moving or resizing a viewport with a native widget type (such as a button or scrollbar). The native widgets may intercept mouse events, so an extra mouse click may be required to finish when dragging the viewport with the mouse. Also, Microsoft Windows does not report mouse events which happen on the window's border. To handle this situation, the Builder allows you to create an offset between the actual mouse position and the position of the control point being moved. You can choose a position in the center of the control point (the default), slightly below and to the right, or above and to the left. Use *Ctrl+Z* to toggle between the three possible values of the offset.

SubDrawing

A subdrawing is used to replicate a template in one drawing or in multiple drawings. When the template is changed, all subdrawings that use the template will change as well. The template may be included in the same drawing or stored in a separate drawing file. By using the subdrawing, you can make a drawing file smaller than it might be otherwise, since only one copy of the template is saved. You can also edit a template in one place to change all of its copies in the drawing.

The subdrawing object can also be used to implement object dynamics, changing the object that is displayed.

There are three types of subdrawing objects that differ in the way they store their template:

- An *included subdrawing* (*SubDrawing From Object* menu option) stores its template in the subdrawing object. 
- A *file subdrawing* (*SubDrawing From File* option) stores the template in an external drawing file. 
- A *palette subdrawing* (*SubDrawing From Palette* option) stores the template in the drawing as a separate palette for convenient editing.

To create an included subdrawing:

1. Select an object to use as a template.

2. Select *Object*, *Properties* to set the attributes of the template object so that its instance will inherit the appropriate settings. Name the object.

3. Click on the  button, or select *Object*, *Create*, *SubDrawing, SubDrawing From Object* menu option, then click in the drawing to select the subdrawing's anchor point.

4. If the template contains several named objects used as icons for object dynamics, enter two colon-separated resource paths, to one of the objects (*ObjectPath*) and its anchor point (*OriginPath*), and press *OK*. To display the whole template, press *OK* without entering *ObjectPath*.

To create a file subdrawing:

1. Click on the  button, or select *Object*, *Create*, *SubDrawing, SubDrawing From File* menu option, then click in the drawing to define the subdrawing's position.

2. Select the drawing file to use. This drawing may contain *$Widget* or *$Drawing* resource to specify the object in the drawing to be used as a template.

3. If the drawing contains several named objects used as icons for object dynamics, enter two colon-separated resource paths, to one of the objects (*ObjectPath*) and its anchor point (*OriginPath*), and press *OK*. To display the whole template drawing, press *OK* without entering *ObjectPath*.

To create a palette subdrawing:

1. Select *Object*, *Create*, *SubDrawing, SubDrawing From Palette* and click in the drawing to define the subdrawing's position.

2. If the palette contains several named objects used as icons for object dynamics, enter two colon-separated resource paths, to one of the objects (*ObjectPath*) and its anchor point (*OriginPath*), and press *OK*. To display the whole palette, press *OK* without entering *ObjectPath*.

3. Edit subdrawing's properties and enter palette object's resource path in the *SourcePath* attribute.

When you create a subdrawing, the Builder places it at the current level of the hierarchy. There are two resources within the subdrawing. The first one, called *Instance*, is a copy of the original template (or of it's subobject if *ObjectPath* is defined) and is also visible in the resource hierarchy under its original name. The second resource, called *Template*, is the original template object. For

containers, the template is drawn directly and both objects refer to the template. For file subdrawings, the *Template* refers to a loaded instance of the subdrawing. This instance is cached and is used by all instances of the subdrawing.

You can use a subdrawing to create a set of objects that refer to the same template, but can be positioned independently. To create additional subdrawing instances, select and copy the first subdrawing object, positioning created copies as desired. All copies will share the same template object. Subdrawing's bindings may be used to assign local values and change behavior of attributes of a particular instance.

If subdrawing objects are used to represent connected nodes, the subdrawing's control point may be used for constraining connecting lines that represent edges.

To edit the subdrawing's template object, use *Traverse, Hierarchy Down*. When finished, use *Traverse, Up* to go back to the top level.

SubWindow

SubWindow is a special type of a subdrawing used to switch drawings displayed in the *SubWindow* object. *SubWindow* has two control points that define an area in which the template drawing is displayed, and its template must be a viewport object.

To create a subwindow:

1. Click on the  button, or select *Object*, *Create*, *SubWindow, SubWindow From File* menu option, then click on two points in the drawing to define the subwindow's position.

2. Select the drawing file to use. This drawing may contain *$Widget* resource to specify the viewport object in the drawing to be used as a template.

3. If the drawing contains several named viewports, enter resource paths to one of the viewports and press *OK*. To display the *$Widget* viewport, press *OK* without entering *ObjectPath*.

A subwindow may be used as a subdrawing with two control points, which is useful for interface objects such as buttons, icons 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.

To edit the subwindow's template object, use *Traverse, Hierarchy Down*. When finished, use *Traverse, Up* to go back to the top level.

Connector

This option creates a connector object which may be used to connect other objects in the drawing. It is useful when implementing node and edge functionality or connecting objects in a diagram.

There are two types of connectors:

- A Recta-Linear connector connects objects with linear segments, maintaining right angles between adjacent segments. There are Horizontal  and Vertical  recta-linear connectors.
- An arc connector connects objects with an arc  .

To create a connector, select the connector type, then click in the drawing to define its shape. For the arc connector, select 3 points to define the arc. For the recta-linear connector, select any number of points to define one or more recta-linear segments and press the Esc key or the middle mouse button to finish.

Series

A series object is a set of dynamically created copies of a **template** object. Typically, you use a series object to create a set of entities with identical, or at least very similar, characteristics, such as the bars in a chart. A series object consists of a template, a **factor** indicating the number of copies, a path along which to arrange them, and a set of generated instances. 

To create a series:

1. Select an object to use as a template.
2. Select *Object*, *Properties* to set the attributes of the template object so that its instances will inherit appropriate characteristics. Name the object.
3. Select *Object, Create*, then select *Line Series* or *Path Series* to create the instance objects.
4. Click on two points to define a line path for a *Line Series*, on which to arrange the series instances. For a *Path Series*, define a transformation to be used for replicating instances.
5. Enter a factor to specify the number of instances to create.

When you create a series object, the Builder names the instances using the template object name and an index; for example, a template named *Rect* with a factor of 3 creates three instances named *Rect0*, *Rect1*, and *Rect2*.

To edit the template object, use *Traverse*, *Hierarchy Down*. When you finish editing, use *Traverse*, *Up* to see the instance objects.

Square Series

A square series is a special type of series object, which presents the instances of its template in rows and columns. The number of rows and columns in the square series determines the number of instances in the series. A square series object consists of a template and a set of generated instances. 

To create a square series:

1. Select an object to use as a template.

2. Select *Object*, *Properties* to set the attributes of the template object so that its instances will inherit appropriate characteristics. Name the object.

3. Select *Object*, *Create*, *Square Series* to create the instance objects.

4. Click on the center point for the square series, and click on two points to define two vectors from the first point; they specify the arrangement of the series instances.

5. When prompted, enter the number of rows, then the number of columns. These values specify the number of instances to create.

When you create a square series object, the Builder names the instances using the template object name and an index. The names are ordered in a simple sequence, even though there are two dimensions to the series. For example, a template named *Rect* with two rows and two columns creates four instances named *Rect0*, *Rect1*, *Rect2*, and *Rect3*.

To edit the template object, use *Traverse*, *Hierarchy Down*. When you finish editing, use *Traverse*, *Up* to see the instance objects.

Polyline

A polyline is a specialized series that can be used to draw line graphs. Like a series object, a polyline consists of templates and a set of instances. For the polyline, the instances are the line and the markers at each point of the line. A polyline can rendered as a single polygon or as a collection of individual segments, depending on the value of the *Segments* attribute.

To create a polyline, click on two points to define the beginning and end of the polyline. The Builder prompts you for the factor, which controls the number of data points along the line.

By default, the polyline has *Marker* and *Polygon* resources that contain the template marker object and the template polygon from which the line segment characteristics are derived. If the *DrawMarkers* and *Segments* attributes are turned on, two groups, *Markers* and *Polygons*, appear among the resources of the polyline. These contain the instances of the template objects.

In order to control a polyline, you must name the control point of the template marker. Use *Traverse, Hierarchy Down* to edit the marker template. This creates a third group of resources within the polyline, called *Points*, which contains the instances of the marker control point. The coordinates of these points control the position of the polyline's points.

Polysurface

A polysurface is a specialized three-dimensional object that can be used to anchor a set of objects along its surfaces. It is used primarily for patching of curved surfaces. Like a square series object, a polysurface consists of a template and a set of instances; for the poly-surface, the instances are polygons arranged in rows and columns.

To create a polysurface, click on a point to define the center of the polysurface, and click on two points to define two vectors from the center point. The Builder prompts you for the number of rows and columns in the surface; these values control the number of surface polygons.

By default, the polysurface has a *Marker*, and a *Polygon* resource, that contain the template marker object and the template polygon from which the instance polygons' characteristics are derived. You only see the instances of the polysurface templates if those objects are named. The template objects of a polysurface are named by default. If the objects are named, two groups, *Markers* and *Polygons*, appear among the resources of the polysurface. These contain the instances of the template objects.

In order to control a polysurface, the control point of the template marker must also be named. The default name is *Point*. Use *Traverse, Hierarchy Down* to edit the marker template if you want to change its name or other characteristics. The instances of the marker control point are found in a third group of resources within the polysurface, called *Points*. The coordinates of these points control the position of the polysurface's points.

Frame

A frame object organizes other objects in a specified arrangement. The points on the frame act as anchors, so other objects can be constrained to them. These are the **frame points**. There are five frame types:

- Point Frame allows anchoring to a single point. Click once to define the point.
- Line Frame allows anchoring to points along a line. Click twice to define the start and end of the line. The Builder prompts you for the factor, which controls the number of anchor points along the line frame.
- 2D Frame allows anchoring to points inside a parallelogram defined by three control points. Click on a point, and click on two points to define two vectors from the first point. The Builder prompts you for the factor, which controls the number of anchor points on each segment of the frame.
- 3D Frame allowing anchoring to points inside a parallel prism defined by four control points. Click on the center point, and click three times to define three vectors for the axes of the frame. The Builder prompts you for the factor, which controls the number of anchor points on each dimension of the frame.
- Free Frame allows anchoring of points to a free-form polygon. Click on each point in the polygon, and click the right mouse button to finish drawing.

Because the control points of the frame coincide with the anchor points, use *Options*, *Show Frame Points* to get access to the anchor points of the frame. You can move the anchor points of the frame by moving its control points.

Edit Toolbox

This option activates the *Edit Toolbox* for fast access to the selected objects' properties. Refer to the *Edit Toolbox* chapter on page 229 for more information.

Properties

Properties displays a dialog that lists the attributes of the selected object and provides access to their transformations, alarms and tags. Note that, within the context of the GLG Toolkit, properties and attributes are synonymous.

The generic properties common for all GLG objects, such as object name or *HasResources*, are displayed at the top of the dialog, and the properties that depend on the type of an object are listed below. For explanations of the generic object properties and attributes of specific object types, see the *GLG Objects* chapter.

The buttons at the bottom of the dialog provide access to adding and editing geometrical dynamics attached to the object, such as move, scale or rotate. The *Custom Props* button at the top of the dialog provides a shortcut for accessing custom properties attached to the object. The selection buttons in the top right corner of the dialog may be used to rotate selection in case when several objects are selected by the mouse.

The Properties Dialogs

When you select an object and then use *Object*, *Properties*, the Builder displays a dialog that shows the properties (attributes) of the object.

A text box on the right side of the attribute row shows the value of the attribute. For attributes that are also objects, the *Selected Object Properties* dialog presents an ellipsis button ... that lets you edit the attribute value using a palette or a menu. Clicking on the ellipsis button displays the *Attribute* dialog; see below.

For attributes of S (String) type that contain multi-line strings, the text box will be read-only and the "[...]" suffix will be shown at the end of the displayed string. An ellipsis button ... can be used to bring the *Attribute* dialog with a text edit box for editing the multi-line string.

If an attribute has dynamics, alarms or tags attached, the *'X', 'A'* and/or *'T'* buttons will be displayed on the right side of the attribute row to provide a quick access to the corresponding transformation, alarm or tag object.

For attributes that are not objects in themselves, the Builder does not present a separate dialog. Such attributes include the *Name*, the *HasResources* flag, and the *Global* attribute.

The contents of the dialog depend on the type of the object (e.g. polygon, group, polyline). Here are some of the object-specific features of the *Selected Object Properties* dialogs:

- For all graphical primitives (such as polygon or text objects) and the viewport object, the *Add/Edit Rendering* button can be used to add and edit rendering attributes of an object, such as gradient or shadows.
- For text objects, the *Add/Edit Text Box* can be used to add a box around the text and edit its attributes.
- For a group object, the dialog includes a set of buttons that give you access to the members of the group without exploding the group. These buttons act as equivalents to the *Traverse* Menu options *Select Next*, *Select Bottom*, and *Edit All* and the *Arrange* Menu options *Add Object to Group* and *Remove Object from Group*.

- For a viewport object, the *More* button provides an access to the attributes of the screen object, which can be thought of as a second set of attributes for the viewport object. Click on the *Back* button to return to the viewport properties. See page 331 for information on font handling for a viewport. The *Add/Edit Light* button adds a light object and lists its attributes.
- For a screen object, the *Add/Edit Fonttable* button allows to define a custom font table and edit its list of fonts. The *Add/Edit Colortable* button allows to add and edit attributes of a custom colortable.
- For the font table, the *Save Font Table* and *Load Font Table* buttons save or load the fonttable object from a file. The *Mark Font Table* button marks the font table for reuse, while the *Use Marked Table* button replaces the fonttable with the marked fonttable. The *Options, Attribute Clone Type* menu controls constraining of the marked fonttable attributes.
- For rendering and box attributes, as well as fonttable, colortable and light objects, the *Delete* button at the bottom of the attribute list deletes the object. The *Mark* button at the top of the *Properties* dialog stores the object for reuse with the *Edit, Add or Use Marked* Object menu option.
- If the object has public properties defined, a button  in the upper right corner of the dialog will switch the displayed properties between object properties and public properties.

The Attribute Dialog

Some objects are both attributes and objects. For these objects, the *Selected Object Properties* dialog for the object includes ellipsis buttons  that let you edit the attribute objects. Clicking on the button displays the *Attribute* dialog.

The *Attribute* dialog displays the following attributes:

- The *HasResources* attribute, which control the position of the attribute's attributes in the resource hierarchy.
- The *Global* attribute, which controls the behavior of the attribute during cloning. This attribute changes its label depending on its current setting; the default label is *Local*.
- A list of values or a palette for setting the value of the attribute. For text strings, it also contains a text edit box for entering multi-line text.
- A set of buttons on the right of the palette for manipulating the attribute object, which are grayed if they are not relevant to the attribute. The *Add/Edit Alarm* button at the bottom can be used to add or edit an alarm to monitor the value of the attribute.
- A *Value* entry field for changing the attribute's value. The value is a double-precision number for D attributes, a triplet of numbers for G attributes, and a text string for S attributes. If the text attribute contains a multi-line string, the Value field is read-only and the text edit box should be used to edit the string.
- An *XfValue* field showing the final transformed value after any transformations attached to the attribute are applied.

The *Dynamics* buttons at the bottom of the *Attribute* dialog provide the only method of attaching a transformation to the attribute. The *Add/Edit Tag* button at the top of the dialog can be used to add a tag for database connectivity.

The Control Point Dialog

You edit a control point's attributes by *Shift*+clicking over the point. The *Control Point* dialog is similar to the *Attribute* dialog. However, it includes a set of arrow buttons for positioning the point, and a *Position* text entry box showing the position of the control point after all transformations have been applied. This field may also be used for setting its position using different coordinate systems. You can reposition the point by typing in the *Value* or *Position* text entry box, or clicking on the arrow buttons.

Public Properties

Public Properties display user-defined public properties of an object. Public properties are used to create components that may be easily edited in the GLG HMI Configurator. The OEM version of the GLG Graphics Builder is used to define an object's public properties. In the non-OEM Builder, the Public Properties menu option can be used to browse public properties of an object.

The *Public Properties* dialog displays public properties in a way similar to the way attributes are displayed in the *Properties* dialog. The only difference is the absence of the *'X'* button for the public properties that have dynamics attached. Instead, the '*x*' character is displayed in the property's ellipsis button ... to indicate the presence of a transformation attached to the object. A scrollbar to scroll the list of properties is automatically activated if required.

Each property has an ellipsis button ... that brings the *Property Object* dialog for editing the property value using an appropriate palette or a menu. The *Property Object* dialog is the same as the *Attribute* dialog described in the previous section.

A button  in the upper right corner of the dialog may be used to switch display between public properties and object properties.

Resources

Resources displays a *Resource Browser* that shows resources of the selected object. If no object is selected, resources of the whole drawing (at the current level of the hierarchy) are shown. Selecting a different object updates the Resource Browser to display the object's resources.

For a discussion of the basics of object resources, see the *Structure of a GLG Drawing* chapter.

The Resource Browser Dialog

The *Resource Browser* dialog lists resources of the drawing. Clicking on any resource entry activates dialogs for editing attributes of the selected resource.

All resources are organized hierarchically, in a way similar to a file and directory structure. You can navigate between the levels by double-clicking on the entries.

Composite resources that contain other resources are annotated with the >> suffix after a resource name. Double-clicking on such a resource enters another level of hierarchy, listing all resources inside the selected composite resource.

The following special entries may also be present in the *Resource Browser* dialog and may be used for navigation:

- / represents the top level viewport (Drawing Area of the Builder).
- . represents the object which is currently selected in the Builder (if any).
- ~ represents the viewport with the editing focus, if it is different from the Drawing Area.
- .. represents the previous level (relatively to the currently selected resource).

The resource browser also provides three toggles which can be used to control what resources are displayed in the browser: *named resources*, *default resources*, *aliases* or any combination of them. By default, all three resource categories are displayed.

For a discussion of the basics of object resources, see the *Structure of a GLG Drawing* chapter.

Tags

Tags brings a *Tag Browser* that shows a list of tags of the selected object, or of the whole drawing if no object is selected. Selecting a different object updates the *Tag Browser* to display the object's tags.

The Tag Browser Dialog

The *Tag Browser* dialog displays a list of tags defined in the drawing or the selected object. Tags are global and have a flat hierarchy, therefore all tags of either the whole drawing or the selected object will be listed in the tag browser. Each tag entry shows the tag's *TagName* and *TagSource* attributes separated by the *'/'* character.

Tags are attached to the data resources to enable an application to access data via tags. When a tag entry in the tag browser is selected with the mouse, two dialogs are displayed: one to edit the attributes of the tag object and another to edit the resource object the tag is attached to.

The *Sort by* toggle of the tag browser may be used to sort the tags by the value of their *TagName* or *TagSource* attributes. The *Filter* field may be used to display only a subset of tags matching a regular expression that may contain the *?* (any character) and * (any sequence of characters) wild cards. The regular expression will be applied to either the tag names or tag sources as controlled by the *Source/Names* toggle on the right side of the *Filter* field. The toggle also controls the *Selection* field, allowing the user to select a tag by typing its *TagName* or *TagSource*.

The *Display Both/One* toggle switches the view to display both the *TagName* and *TagSource*, or just one of them based on the current sorting setting. If tags are sorted by the tag name, the *TagName* is shown in the *Display One* mode. If tags are sorted by the tag source, the *TagSource* is displayed.

The *Unique Tag Sources/Names* toggle controls the display of tags with identical tag sources (when sorting by tag sources) or tag names (when sorting by tag names). By default, the toggle is unchecked and all instances of tags with the same tag source or tag name will be displayed in the tag browser. If the toggle is checked, only the first instances of tags with the same tag source or tag name will be displayed.

To add a tag, click on the *Add Tag* button in the *Attribute* or *Resource Object* dialog. To delete a tag, click on the *Delete* button in the *Data Tag* dialog.

For a discussion of the basics of attribute tags, see the *Tags for Database Connectivity* chapter on page 31 and the *Tag-Based Data Access and Database Connectivity* chapter on page 60.

Alarms

Alarms opens an *AlarmBrowser* that shows a list of alarms attached to the selected object or all alarms of the whole drawing if no object is selected. Selecting a different object updates the *Alarm Browser* to display the object's alarms.

The Alarm Browser Dialog

The *Alarm Browser* dialog displays a list of alarms attached to the attributes of the selected object to monitor their values. If no object is selected, all alarms defined in the whole drawing will be displayed. Each alarm entry shows the alarm's *AlarmLabel* attribute.

Alarms are attached to object attributes to monitor their values. When an alarm entry in the alarm browser is selected with the mouse, two dialogs are displayed: one to edit the attributes of the alarm object and another to edit the attribute the alarm is attached to.

The *Filter* field may be used to display only a subset of alarms whose *AlarmLabel* matches a regular expression that may contain the *?* (any character) and * (any sequence of characters) wild cards. The *Selection* field may be used to select an alarm by typing its *AlarmLabel*.

For a discussion of the basics of attribute alarms, see the *Alarms* chapter on page 39 and the *Integrated Alarms for Value Monitoring* chapter on page 63.

Object Dynamics

The *Object Dynamics* submenu provides options for adding and editing geometrical dynamics.

Add Dynamics

Add Dynamics adds geometrical dynamics (such as move, scale or rotate) to the selected object. The dynamics are attached in the form of a dynamic transformation.

For an explanation of the difference between "static" and "dynamic" and a description of the possible transformations, see the *Transformation Object* chapter on page 151.

You can animate a drawing by attaching a dynamic transformation to a named object, and naming the factor for the transformation. (Entering a name in the *Variable Name* field in the *Add Dynamics* dialog assigns it to the *Factor* attribute in the *Edit Dynamics* dialog.) When you use *Run*, *Start* to execute the animation, specify the named factor in the command line; see page 354.

To add dynamics to object attributes (such as visibility or color dynamics), use the *Add Dynamics* button of the *Attribute* dialog. Refer to the *Adding Attribute Dynamics* chapter on page 241 for details.

The Add Dynamics Dialog

The *Add Dynamics* dialog lets you define a parametric (dynamic) transformation to attach to the selected object.

To specify the transformation, use the *Transformation Type* option, which lists the geometrical transformations. See the *GLG Objects* chapter for specifications of these transformations. When you select a transformation, the content of the dialog changes, providing appropriate text entry boxes and buttons for defining the transformation.

To switch from defining a dynamic transformation to transforming points or defining a static transformation, use the *Action* option.

Some of the text entry boxes are paired with buttons; for these values, you can specify these values by typing values or by clicking with the mouse. Clicking on a button and then in the drawing area records the mouse position in the corresponding text entry box. For example, clicking on the *Distance X In Drawing* button and clicking on two points in the drawing calculates and records the distance between the points in the X direction. When using buttons for defining transformation's geometry in the drawing, notice the prompt at the bottom of the drawing area that provides information on the number of points to select with the mouse.

For most transformations, a *Reverse* button provides a way to reverse a transformation by inverting its parameters.

Use the *Variable Name* text entry box to name the controlling parameter of the dynamic transformation. The range parameters may be used to map the range of the input data to the range of the required change of the transformation's factor. If the default range parameters are modified, a *Range Conversion* transformation will be attached to the dynamics' *Factor* and the name entered in *Variable Name* will be assigned to the *Input Value* of the *Range Conversion* transformation.

For the *Path* transformation, the dialog provides several ways of defining the path points. The *Points In Drawing* button can be used to define the path points with the mouse. *Use Object* uses points of an object (such as polygon, arc, or spline) selected with the mouse. *Constrain To Polygon* constrains points of the path to the points of the selected polygon; if polygon's points change their position, the path changes accordingly. However, adding or deleting points from the polygon after using the *Constrain To Polygon* button does not affect the path. Finally, *Select Path* can be used to select the path object ID, so that adding or deleting points from the object is reflected in the path. *Select Path* can also be used to use objects other than polygons, such as arcs, connectors or splines.

For the *Use Marked* transformation type option, the dialog provides the *Clone Type* option that controls if the parameters of the added transformation copy are constrained to the corresponding parameters of the original transformation.

Use the *Apply* button when you have finished specifying the transformation.

Edit Dynamics

Edit Dynamics lets you change the values of a dynamic transformation.

This option is equivalent to selecting the *Edit Dynamics* button from the *Properties* dialog. It is grayed when the selected object has no dynamic transformations.

The Edit Dynamics Dialog

The *Edit Dynamics* dialog lets you change parameters of dynamic transformations attached to the object, change their order and delete selected transformations from any position in the list.

A list of transformations attached to the object is displayed on the left side of the dialog. Selecting a transformation in the list displays its properties.

For stock transformations, the attributes are listed in the dialog, the same way as object attributes are listed in the *Properties* dialog described on page 339.

For predefined transformations, their public properties are listed, the same way an object's public properties are listed in the *Public Properties* dialog described on page 341. A button  in the upper right corner of the dialog may be used to switch display between the public properties and the full display of the predefined dynamics.

Each attribute or property row has an ellipsis button  that provides access to editing the attribute or property, as well as adding tags, alarms and second-level transformations to it. For convenience of editing, the dialog also lists a name of each attribute or public property.

The *Up* and *Down* buttons on the right of the transformation list allow the user to change the order of the transformations by moving the selected transformation up or down the list. The *Mark Object* and *Mark List* buttons may be used to mark the dynamics for reuse. The *Mark Object* button marks the selected transformation, while *Mark List* marks the whole list in case if it has more than one transformation. To reuse the marked dynamics, use the *Use Marked* option when adding dynamics to an object.

The *Back* button may be used to return from recursive editing of the second-level transformations attached to the attributes of the main transformation. When editing transformations attached to the attributes of the main transformation, the title of the *Edit Dynamics* dialog displays the level of the transformation being edited.

Delete Dynamics

The *Delete Dynamics* submenu provides options to let you remove static or dynamic transformations from the object or it's subobjects.

This option is grayed when neither the selected object no its subobjects have transformations attached.

Delete Object's Transformation

Delete Object's Transformation removes either the first or last transformation in the list from the selected object. If you are using the object's coordinates to view the drawing (see page 312), the last transformation in the list is deleted. Otherwise, the first transformation in the list is deleted. Note that this order applies even if the *Edit Dynamics* dialog is open and a different transformation is selected.

Delete Sub-Object's Transformation

For a group or other composite object, *Delete Sub-Object's Transformation* removes a transformation from an individual object in the group.

Transform Points

Transform Points changes the coordinates of the control points of the selected object.

Transforming an object's points changes the object permanently by changing the coordinates that define its points. This kind of transformation is not saved as part of the object, but is applied immediately. Therefore, it will not appear in the object's transformation list.

If the object's *MoveMode* is set to STICKY MODE, this operation also changes the control points of any geometrical transformations attached to the object. If the object's *MoveMode* is set to MOVE BY XFORM, the operation adds a static transformation to the object instead of changing its coordinates, see page 69.

The Transform Points Dialog

The *Transform Points* dialog lets you define the transformation to apply to the selected object's points. Keep in mind that transforming an object's points irrevocably changes the object; such a transformation cannot be edited or deleted.

To specify the transformation, use the *Transformation Type* option, which lists the transformations for geometric data. See the *GLG Objects* chapter for specifications of these transformations. When you select a transformation, the content of the dialog changes, providing appropriate text entry boxes and buttons for defining the transformation, which are the same as in the *Add Dynamics* dialog described on page 343.

To switch from transforming points to defining a static or dynamic transformation, use the *Action* option.

For most transformations, a *Reverse* button provides a way to "undo" a transformation. It inverts the parameters of the last transformation you applied, which has the effect of restoring the object to its untransformed state when the inverse transformation is applied.

The *Variable Name*, *InLow*, *InHigh*, *OutLow*, and *OutHigh* options are not available; these options apply only to dynamic transformations.

Use the *Apply* button when you have finished specifying the transformation.

Add Static Transformation

Add Static Transformation creates and applies a matrix (static) transformation to the selected object. The transformation is saved as part of the object. You can delete a static transformation using *Object*, *Delete Dynamics*; see page 344.

A matrix transformation cannot be edited directly. If several matrix transformations are applied, they are merged in a single matrix transformation that has a combined effect of all applied matrix transformations. Therefore, the only way to change a matrix transformation is to modify its matrix by applying another matrix transformation.

To see the object without transformations, use *Traverse*, *Transformation Down*; see page 314. For an explanation of the difference between "static" and "dynamic" and a description of the possible transformations, see the *GLG Objects* chapter.

The Matrix Transformation Dialog

The *Matrix Transformation* dialog lets you define a matrix (static) transformation to attach to the selected object. Keep in mind that a matrix transformation cannot be edited directly.

To specify the transformation, use the *Transformation Type* option, which lists the transformations for geometric data. See the *GLG Objects* chapter for specifications of these transformations. When you select a transformation, the content of the dialog changes, providing appropriate text entry boxes and buttons for defining the transformation, which are the same as in the *Add Dynamics* dialog described on page 343.

To switch from defining a static transformation to transforming points or defining a dynamic transformation, use the *Action* option.

For most of the transformations, a *Reverse* button provides a way to "undo" a transformation. It inverts the parameters of the last transformation you applied, which has the effect of negating the effect of the previously applied static transformation when the inverted transformation is applied.

The *Variable Name*, *InLow*, *InHigh*, *OutLow*, and *OutHigh* options are not available; these options apply only to dynamic transformations.

Use the *Apply* button when you have finished specifying the transformation.

Tooltip

The *Tooltip* submenu provides options for adding and editing object tooltips.

Add Tooltip

Add Tooltip attaches a tooltip action to the selected object and opens the action's Properties dialog for entering the tooltip string as a value of the action's *Tooltip* attribute.

To activate tooltip processing, the *ProcessMouse* attribute of the viewport containing the object (or a parent viewport) has to include the *Tooltip* mask. The *Run* mode of the Builder may be used to test the tooltips.

Edit Tooltip

Edit Tooltip opens the Properties dialog for editing the Tooltip action attached to the selected object.

Use the *Object, Custom Properties, Edit Custom Properties* menu option to edit old-style (prior to v. 3.5) tooltips defined as named *TooltipString* resources.

Delete Tooltip

Delete Tooltip deletes a tooltip action attached to the selected object.

Use the *Object, Custom Properties, Edit Custom Properties* menu option to delete old-style (prior to v. 3.5) tooltips defined as named *TooltipString* resources.

Actions

The *Actions* submenu provides options for adding or editing actions and commands attached to objects in the drawing.

Add Command

Add Command attaches a command action to the selected object. It prompts the user to select a type of command from a list of available commands, then opens a dialog for specifying values of the command's parameters.

The *Back* button may be used to return to the action's Properties dialog after the command data has been entered. The action properties specify the event that triggers the command's execution. Refer to the *Action Object* chapter on page 176 for more information.

Add Custom Mouse Event

Add Custom Mouse Event attaches a custom event action to the selected object and opens the Properties dialog for editing the action's attributes. The action's properties specify the type of the mouse event that triggers the custom event. Refer to the *Action Object* chapter on page 176 for more information.

Add Mouse Feedback

Add Mouse Feedback attaches a mouse feedback action to the selected object and opens the Properties dialog for editing the action's attributes. The action's properties specify the type of the feedback, as well as the mouse event that triggers it. Refer to the *Action Object* chapter on page 176 for more information.

Add Input Command

Add Input Command attaches an input command action to the selected input object. It uses the same user interface as the *Add Command* option described above, but is active only when an input object, such as a button, a toggle or a slider, is selected.

Add Input Action

Add Input Action attaches an input action, such as a custom event, to the selected input object. It uses the same user interface as the *Add Custom Event* option described above, but is active only when an input object, such as a button, a toggle or a slider, is selected.

Edit Actions

Edit Actions opens an Action List dialog for editing actions attached to the selected object. The dialog provides an interface for editing attributes, reordering or deleting individual actions. Refer to the *Action Object* chapter on page 176 for more information.

The *Actions* button in the *Status Panel* at the bottom of the drawing area may be used as a shortcut.

Delete All Actions

Delete All Actions deletes all actions attached to the selected object. To delete individual actions, use the *Edit Actions* option, select an action to be deleted, then press the *Delete* button.

Mark Actions

Mark Actions stores a copy of the selected object's actions in the clipboard to subsequently add them to other objects in the drawing. To select individual actions instead of the list of all actions attached to the object, use the *Mark Object* button in the action's properties dialog.

Add Marked Actions

Add Marked Actions adds a copy of the previously marked actions to the selected object.

Add Tooltip (3.4)
Add MouseClick Event (3.4)
Add MouseOver Event (3.4)
Edit/Delete Tooltip Or Event (3.4)

Options for adding or editing the old-style (prior to v. 3.5) tooltips and custom events. These options are disabled by default, but may be enabled by setting the value of the *DisablePre3-5Menus* property to 0 in the *glg_config* file.

Custom Properties

This button brings the Custom Properties sub-menu with the options listed below.

Add Custom Property

Add Custom Property attaches a custom property to an object. Custom properties are saved with the drawing and may be used to associate application-specific data with objects. Custom properties other then data may be attached to objects programmatically using the Extended API.

The presence of custom properties is indicated by the *Data* indicator in the *Status Panel* when an object with custom properties is selected.

To edit custom properties, select the object to which the custom properties are attached and use *Object*, *Edit Properties*, or click on the *Data* indicator in the *Status Panel*. Alternatively, display the object's *Properties* dialog and click on the *Custom Props* button.

Add D Property

Creates a D data property for holding a double-precision value.

Add S Property

Creates a G data property for holding a triplet of XYZ or RGB values.

Add G Property

Creates an S data property for holding a text string.

Add D List

Creates a list of custom properties containing a single D data property and adds the list into the objects' list of custom properties. This is different from simply adding a custom property of a D type: selecting *Add D List* creates a list of lists. The new list entry appears as an "*Unnamed >>"* in the list of object's custom properties. The ">>" symbols indicate that the list can be opened for editing by double-clicking on it with the left mouse button. When the list is opened, it can be given a name and its content can be edited as well. To add more D properties to the list, press the *Add* button at the bottom of the list dialog. To edit lists' elements, select an element and edit it using the displayed *Attribute* dialog. To get back to the object's custom properties list, double-click on the ".." entry.

Add S List

Same ad *Add D List*, but adds a list of S properties.

Add G List

Same ad *Add D List*, but adds a list of G properties.

Add Predefined Set

Adds the selected predefined custom property set from a list of available custom data sets. Refer to the *Custom Data Sets and Custom Commands* section on page 287 for information on customizing the Builder to add new custom data sets.

Add Marked List

Adds a marked list of custom properties to object's custom property list. This is different from *Add Marked Properties* below which adds all properties of the list, but not the list itself. A custom property list may be marked by pressing the *Mark List* button in the list editing dialog.

Edit Custom Properties

Edit Custom Properties lets you change the names and values of custom properties associated with the object, as well as delete selected custom properties from a list.

When you use the *Edit Custom Properties* option, the Builder displays a list of custom properties attached to the object. Selecting a property in the list brings up the *Attribute* dialog for editing it.

The *List Name* field can be used to name the custom property list. The *HasResources* toggle controls the corresponding flag of the list, which may be used to make the named properties of the list to appear as resources of the list instead of resources of the object the list is attached to. The *Mark List* buttons marks the list for reuse with the *Object, Custom Properties, Add Marked List* menu option.

The *">>"* suffix is displayed if a property itself is a list that contains other properties. Double-clicking on such a property opens it for editing and displays a list of properties it contains. When editing a list property, *List Name*, *HasResources* and *Mark List* control the attributes of the list property itself instead of the attributes of the custom property list attached to an object.

The buttons at the bottom of the *Custom Properties* dialog may be used to delete or add properties to the list. The *Delete* button deletes the currently selected property, while the *Add* button adds a new property to the list.

An option menu to the right of the *Add* button defines the type of the custom property to add. By default (*Default* option), it adds a property of the same type as the currently selected property, inheriting the selected property's name and value (change the name of the newly added property to avoid name conflicts). The option menu also has options for adding new custom properties of D, S and G data types, as well as adding lists of properties containing D, S and G properties inside them. The last option (*Marked*) may be used to add a previously marked list of properties.

When a list of properties is added, double-click on it to get access to the properties inside it. The added list becomes the sublist of the object's custom property list. The sublist may contain other sublists inside, with no restrictions on the sublist depth. When editing sublists, the level of the sublist is displayed in the title of the *Custom Properties* dialog.

NOTE: Programmatically attached custom properties other then data properties can't be edited in the Builder.

Delete All Custom Properties

Delete All Aliases destroys all custom properties attached to the object. The *Delete* button of the *Edit Custom Properties* dialog may be used to delete individual properties.

Mark Custom Properties

Marks all custom properties attached to the selected object.

Add Marked Properties

Adds custom properties marked with *Mark Custom Properties* to the custom property list of the selected object.

Aliases

This button pops up the *Aliases* sub-menu with the options listed below.

Add Alias

Add Alias creates an alias object.

An alias object provides a way to assign logical names to arbitrary resource hierarchies. The resource can then be accessed using the alias instead of the hierarchical resource path.

An alias object is not visible and does not appear in the drawing. Its presence is indicated by the *Alias* indicator in the *Status Panel* when an object with aliases is selected.

To edit an alias, select the object to which the alias is attached and use *Object*, *Edit Aliases*, or click on the *Alias* indicator in the *Status Panel.*

The Alias List Dialog

Adding an alias opens an *Alias List* dialog. The left part of the dialog contains a list of aliases attached to the object; alias properties of the selected alias are displayed on the right.

A new alias is created with both *Alias* and *Path* attributes set to "Alias". Change the *Alias* attribute to the alias name you want to assign, then enter the resource path to be aliased in the *Path* attribute, or press the ellipsis button  next to the *Path* attribute to define the path using the *Resource Browser.*

The *Up* and *Down* buttons to the right of the alias list may be used to reorder aliases in the list when an object has more than one alias attached. The *Name* field at the top of the dialog may be used to define a name of the alias object itself, in case if an application needs to access the alias object programmatically.

The *Mark Object* and *Mark List* buttons in the *Alias List* dialog may be used to mark the currently selected alias object or the whole alias list for reuse. To add marked aliases to a different object, select the object and use *Object, Aliases, Add Marked Aliases* from the main menu.

To delete the currently selected alias from the alias list, use the *Delete* button at the bottom of the dialog.

Edit Aliases

Edit Aliases lets you change the attributes of an alias after it has been added, as well as delete selected alias from a list.

When the *Edit Alias* option is used, the Builder displays a list of aliases attached to the object. Selecting an alias in the list shows its parameters. The *Alias* parameter specifies the logical name to use, and the *Path* parameter specifies the resource path. The ellipsis button  next to *Path* activates the Resource Browser for selecting the resource. Refer to the previous section for a detailed description of the *Alias List* dialog.

Delete All Aliases

Delete All Aliases destroys all aliases attached to the object. The *Delete* button of the *Edit Aliases* dialog may be used to delete individual alias objects from a list.

Mark Aliases

Marks all aliases attached to the selected object.

Add Marked Aliases

Adds aliases marked with *Mark Aliases* to the alias list of the selected object.

History

This button pops up the History sub-menu with the options listed below.

Add History

Add History attaches a history object to the selected object.

A history object provides scrolling functionality, such as scrolling behavior of GLG graphs. History provides a mechanism to animate resources that use sequential numbers in their resource names. It provides an access to these resources via a single entry point and scrolls resource values according to the specified scrolling type.

For example, a history object may be attached to a group that contains a set of rectangles named *Rectangle0*, *Rectangle1*, *Rectangle2* and *Rectangle3*. It can also be used with series objects, since series name their instances by adding a sequential number at the end of each instance's name. For example, if a template of a series object is named *DataSample*, the series' instances will be named *DataSample0, DataSample1, DataSample2* and so on.

A history object is not visible; it does not appear in the drawing and only appears in the resource hierarchy if you explicitly name it. Its presence is generally indicated by a resource named *EntryPoint* in the resource hierarchy and by the *History* indicator in the *Status Panel* when the object is selected. To animate the object containing the history object, you provide input data to the *EntryPoint* resource.

To edit a history object, select the object the history is attached to and use *Object*, *Edit History*, or click on the *History* button in the *Status Panel*.

The Add History Dialog

The *Add History* dialog lets you specify the resource to animate. It prompts you to enter the resource name that will be used as a template for accessing all sequential resources controlled by the history object.

Specify the constant part of the resource name, substituting a percent sign (%) for the variable part of the name. You can specify a "path" to a child object, using forward slashes (/) as you would in a UNIX path specification.

For example, for a series named *Series1* with *HasResources=YES* and a template named *Triangle*, the instances are named *Triangle0*, *Triangle1*, *Triangle2*, and *Triangle3*. To animate the fill color, select *Series1* and use *Object*, *Add History*. In the *Add History* dialog, specify *Triangle%/FillColor* in the *Resource Name* text entry box. The resulting *EntryPoint* object is defined at the same level as the *Triangle* object (*$Widget/Series1/EntryPoint*).

The entered resource path is used as a template to access all sequential resources controlled by the history. The location of the percent sign in the resource path tells the history object where to add sequential numbers to form sequential resource names. The numbers always start with 0. If the percent sign is omitted, the history will add numbers at the end of the resource path template.

Edit History

Edit History lets you change the attributes of a history object, as well as delete selected history objects from the list of attached history objects.

When you use the *Edit History* option, the Builder displays the *History List* dialog with a list of history objects attached to the selected object. Selecting a history object in the list shows its attributes, so you can edit them. Refer to the *History* section on page 139 for information on the *History* object's attributes.

Delete All Histories

Delete All Histories destroys all attached history objects. The *Delete* button of the *Edit History* dialog may be used to delete individual history objects from the list.

If you explode an object with a history object, the group that remains after exploding it retains the history object. Exploding that group discards the history object.

Run

The *Run* Menu provides options to let you test the animation of drawings. The simplest way to test a drawing is by using the test data generated by the GLG *datagen* program.

Start

Start invokes a process to animate a drawing. The process executes the command or commands given it in the dialog presented when you select *Start*. The default animation program is *$datagen*, a test program that is part of the GLG Toolkit; see the *GLG Programming Tools and Utilities* chapter.

To animate a drawing, use *Run*, *Start*, specifying a named resource as part of the command.

For example, to animate an object named *valve1* with a dynamic transformation that has a named factor (variable name) of *rotate*, use this command line:

```
$datagen -sin d 0 1 valve1/rotate
```

The *$datagen* is a GLG shortcut that prevents the initiation of another process and uses an internal data generator. See the *The Data Generation Utility* section of the *GLG Programming Tools and Utilities* chapter for more information on data generation options.

If no command is given the run process, or if an error is encountered when trying to execute the command, the Builder searches the drawing for a string resource called *"$DatagenString"*, and uses it as a run command.

When a new drawing containing *"$DatagenString"* resource is loaded, the resource value is used to initialize the animation command. The *Store Run Command* option may be used to store the current animation command in the existing *"$DatagenString"* resource of the drawing.

In the Run mode, the Builder activates the *Run Toolbar* that contains the following controls:

- **Updates** - displays the number of updates performed from the start of the Run mode.
- **Seconds** - shows the number of seconds the Run mode was active; this does not include the time when the Run mode was paused.
- **Updates/sec** - displays an average number of updates per seconds. Clicking on this control changes it to **Secs/Update**, which displays an average number of seconds it takes to execute one update.
- **Pause / Resume button** - temporarily pauses drawing updates without quitting the Run mode. Pressing the button the second time resumes updates.
- **Stop button** - stops the Run mode.
- **Update Speed** - controls the frequency of updates, from 9 (maximum) to 1 (minimum).

Many of the Builder's menu options and buttons become unavailable in the Run mode. To terminate the animation, use *Run*, *Stop*.

Stop

Stop halts the animation of a drawing, killing the process started with the *Start* command.

To restore the original appearance of the drawing, use *File*, *Reset Drawing*.

Restore Values on Stop

Restore Values on Stop controls the behavior of the data generator. If the option is checked, the resource values animated with the data generator will be restored to the values they had before animation started.

This option does not restore the state of the graph's history: reset the drawing using *File, Reset* to reset the graphs.

Store Run Command

Store Run Command stores the current animation command in the existing *"$DatagenString"* resource of the drawing. The value of the resource is used to initialize the default animation command when a drawing containing *"$DatagenString"* resource is loaded.

To create a *"$DatagenString"* resource in the drawing, create a text object outside of the widget at the top of the drawing hierarchy, and name it's String resource *"$DatagenString"*.

Options

The *Options* Menu provides options to let you customize the operation of the Builder. The *glg_config* Builder configuration file can be used to specify default values of various options.

Draw Grid

The *Draw Grid* submenu provides options to let you display grid lines with a drawing. The grid is displayed at a selected grid interval in the GLG world coordinates. In addition to the constant grid spacing options, two adaptive options are provided. The *Adaptive (Constant Space)* option automatically selects a grid interval to maintain approximately constant visual grid spacing, while *Adaptive (Constant Number)* maintains a constant number of grid lines. The *Custom* option allows the user to specify a custom grid interval in world coordinates. To disable grid, use the *No Grid* option.

The grid is not saved as part of the drawing. It is just a convenience for editing a drawing. To save the grid as part of the drawing, specifying a value for the viewport's *GridValue* attribute.

To position objects relative to the grid, use *Options*, *Snap To*, *Grid*.

Snap To

The *Snap To* submenu provides options to let you align objects. Snapping places objects by rounding their coordinates as you draw them.

The options on the *Snap To* submenu specify the resolution of the alignment grid in the GLG world coordinates. If you specify *Snap to Grid*, the Builder snaps to the grid and also to the midpoints of the grid. The *0 (No snapping)* option is used to disable snapping, and *1 (Round to integer)* option is used to get rid of the fractional coordinate values, which is almost the same as no snapping, but with integer coordinates. The *Custom* option allows the user to specify a custom rounding value.

The *Snap To* submenu only affects the placement of points and objects with the mouse. Other means of specifying points, such as entering coordinate values or using existing values, are not affected by the *Snap To* option.

Show Axis

Show Axis shows or hides the axis icon. The axis icon indicates the center and orientation of the current view and coordinate system.

To change the view projection and the coordinate system used to view the drawing, use the options on the *View* menu.

Show Coordinates

Show Coordinates shows or hides the mouse coordinate display window in the lower right corner of the Builder. Both the screen (S) and world (W) coordinates in the selected coordinate system are displayed. To change coordinate system, use *View, Coordinate System*.

Show Default Span

Show Default Span shows or hides red markers that display the extent of the drawing area in the unzoomed state.

Save Format

Save Format determines the format the Builder uses to save a drawing to a file. The Builder can save files in three different formats:

- Binary, which loads quickly but is not portable across platforms that use different binary data representations.
- ASCII, which is completely portable across platforms, but loads slightly slower than binary.
- Extended, which is portable across platforms and across versions of the Builder, but loads most slowly. This format is used only to import drawings to the older versions of the Toolkit and is available only in the Enterprise Edition of the Graphics Builder.

The default format for the saved drawing is ASCII. This is the recommended format that works not only across all hardware platforms, but also binary (C/C++/ActiveX), Java and C#/.NET applications.

Save Compressed

Save Compressed enables or disables saved drawing compression. When the drawing compression is enabled, drawings are saved using gzip-compression. The saved file extension is not changed when the drawing is saved compressed. This option controls only saving operations: on loading, the drawing format is recognized and handled automatically. The editor can also load drawings compressed outside of the editor using gzip utility.

The default is Save Compressed. Disable the drawing compression if the drawing will be used for code generation.

Selection Options

Disable Dynamics For Editing

Disable Dynamics For Editing disables geometrical dynamics (such as move, scale or rotate) when the object is selected. When the option is set, the object will appear in its untransformed state with dynamics disabled. This is convenient for editing such objects as needles of dial controls, so that the editing is performed in the object's initial state, with the needle pointing along the X axis. This makes it easier to stretch the object in X or Y direction, as well as enter coordinate values or use snapping to the grid.

The option disables only the geometrical dynamics attached to the object. It does not disable dynamics attached to the object's control points.

Selection Display

Selection Display toggles between three modes of displaying the selection:

- *Resize and Reshape* displays both the object's resize box and the object's control points. This is the default mode.
- *Reshape* shows only the control points.
- *Resize* shows only the resize box points. This mode may be used to speed up editing groups with a lot of control points.

Control Points Display

The *Control Points Display* toggles between three modes for displaying control points:

- *Object's points* display only control points of drawable objects (polygons, circles, etc.).
- *Object's and dynamics' points* display control points of drawable objects and control points of dynamics (*Start* and *End Points*, *Rotate* and *Scale Centers*, etc.). This option helps to visualize dynamics attached to objects.
- *Object's, dynamics' and point dynamics' points* is similar to *Object's and dynamics' points,* but also displays control points of dynamics attached to the control points.

Reference Resizing

This option controls the reference resizing modes when the reference object is stretched using the resize box:

- *Resize Box* mode resizes relative to the box. This is the general way objects are resized.
- *Around Point* mode uses the control point of the reference object as the center of scaling, making sure the control point position does not change when the reference is resized. This may be used to resize reference objects used as nodes: the node will resize without changing the position of its attachment point.

Show Frame Points

For a selected frame object, *Show Frame Points* toggles between its control points and its frame points. Only one of the two sets of points is available to be selected at any time, because several of these points may coincide on a frame (all of them for a free frame or a point frame).

This option also affects the recta-linear connector object, displaying the control or constrained points.

If you want to move or resize a frame or connector object, use their control points. If you need to constrain other objects to the frame or to the middle point of a recta-linear path, they should be constrained to the constrained points.

Edit Action Data as List

Edit Action Data as List toggles the way action and command data of the SEND_COMMAND and SEND_EVENT actions are displayed for editing: as a Public Properties dialog, or a Custom Properties dialog that allows individual properties to be added or removed. A button  in the upper right corner of the Action Properties dialog provides a convenient shortcut.

Enable ConstrainOne

Enable ConstrainOne enables or disables the *Constrain One* button in the Attribute Dialog. The *Constrain One* button is disabled by default and may be enabled for advanced use. The *EnableConstrainOne* option in the *glg_config* file may be used to define the default setting for the Graphics Builder.

Trace Attribute Constraints for Mark0

Trace Attribute Constraints for Mark0 enables or disables constraints tracing for the attribute marked as Mark0 using the *Mark, Mark0* buttons in the Attribute or Resource dialogs. Refer to the *Constraints Tracing* section on page 238 for mode information.

Color Options

Swap Color Palettes

Swap Color Palettes toggles the displayed color palette between the default color palette and custom color palette. The color palette may also be swapped by *Ctrl*-clicking on the color palette.

Refer to the *Custom Color Palette* section on page 282 for information on defining a custom color palette.

Pastel Colors

Pastel Colors toggles the displayed color palette between the standard and pastel colors. Pastel colors may also be activated and deactivated by *Shift*-clicking on the color palette.

255 Color Display

255 Color Display toggles the range of the color RGB values between the default [0;1] range and the [0;255] range familiar to the Windows users. The color display range changes only the color RGB display in the Builder. The run-time color RGB values and the values saved in the drawing still use the default [0;1] range.

Dynamics Options

Full Display of Predefined Dynamics

Full Display of Predefined Dynamics controls the way predefined dynamics are displayed. If the option is not checked (default), the Edit Dynamics dialog displays public properties of predefined dynamics for convenient editing. If the option is checked, the dialog displays all attributes of the dynamics for advanced editing.

Show Predefined Dynamics First

Show Predefined Dynamics First controls the choices of dynamics presented in the attribute dynamics menu. If the option is checked (default), the predefined dynamics options are shown first to provide intuitive attribute dynamics, and stock dynamics are shown when the *More* button is pressed. If the option is unchecked, the stock dynamics options are shown first.

Data Browser Options

Browse Tag Names
Browse Tag Sources

Control the usage of the Data Browser. If *Browse Tag Sources* is selected, the tag string selected in the Data Browser is assigned to the *TagSource* attribute of the tag being edited, otherwise it is assigned to the tag's *TagName* attribute.

Attribute Clone Type

Attribute Clone Type defines the clone type to use when multiple copies of an attribute object are added. For example, when a Rendering object is added to all objects in a group using the group's *Edit All* option with the default *Constrained Clone* setting of the clone type, all attributes of rendering objects attached to objects in the group will be constrained. If a rendering attribute of one object is changed, all objects will change. If the clone type is set to *Full Clone*, each object will have its own, unconstrained copy of the rendering attributes. If the clone type is changed to *Strong Clone*, all attributes that are *Global* or *SemiGlobal* will be constrained to the corresponding attributes of the original Rendering object.

Attribute Clone Type controls adding Rendering, Box Attributes, Light, Font Table and Color Table objects to all objects in a group using the *Edit All* option. It also controls constraining of attributes of an attribute transformation when the attribute the transformation is attached to is unconstrained. If the clone type is set to *Constrained Clone*, the attribute itself will be unconstrained, but the attributes of the transformation attached to it will still be constrained. The *Attribute Clone Type* also controls constraints of the transformations added to an object's control points when the object's transformation is exploded using the *Add Copy To Points* option.

Refer to the *GlgCloneObject* method on page 137 of the GLG Programming Reference Manual for more details on the clone types.

Paste Clone Type

Paste Clone Type defines the clone type to use when copies of an object are pasted into the drawing using the *Copy/Paste* or *Clone* operations. For the *Copy/Paste* sequence, the clone type must be set before *Copy.*

The default setting is *Full Clone* and the attributes of the pasted copies are not constrained. If the setting is changed to *Strong Clone*, all attributes of the pasted object that are *Global* or *SemiGlobal* will be constrained to the corresponding attribute of the original object.

Refer to the *GlgCloneObject* method on page 137 of the GLG Programming Reference Manual for more details on the clone types.

Subdrawing Traversal

Subdrawing Traversal controls the interface for loading and saving subdrawings when the subdrawing is entered using the *Hierarchy Down* button and exited using *Hierarchy Up*. By default, it is set to *Verbose (Save Prompt)*, which displays confirmation dialogs when loading and saving the modified subdrawing. The *Silent (Auto-Save)* option eliminates confirmation dialogs to automatically load and save the subdrawing. It may be used to simplify the process for beginner users.

Appearance

Detach Palettes

Detach Palettes frees the button palettes from the main Builder window. Detaching the palettes increases the available area for viewing a drawing. The palettes can be iconized and resized independently of the main Builder window. Detaching the palettes does not affect the content of the drawing.

To reattach the palettes to the Builder window, select *Detach Palettes* again.

Scrollbars

Scrollbars shows or hides the scrollbars in the GLG Graphics Builder window.

Toolbar

Toolbar shows or hides the toolbar at the top of the GLG Graphics Builder window.

Modal Dialogs

This option controls modality of the dialogs. If dialogs aren't modal, clicking in the drawing selects an object and may close the currently displayed dialog. This option enables fast navigation and is intended for expert users.

If dialogs are modal, clicking in the drawing area is allowed only for some operations. This setting helps learning the Builder and is recommended for novice users.

Display OpenGL Info

This option displays diagnostic information about the status of the OpenGL driver. If the driver has been successfully initialized, it also displays the OpenGL renderer and vendor information to assist in troubleshooting the OpenGL driver setup problems.

The *-verbose* command-line option may be used to display extended diagnostic information on Linux/Unix, or to record the information in the *glg_error.log* file on Windows. The location of the file is determined by the GLG_LOG_DIR and GLG_DIR environment variables as described in the *Error Processing* section on page 51 of the *GLG Programming Reference Manual*.

Save Layout

Save Layout saves the drawing that defines the layout and appearance of the of the Builder. This drawing can be edited like any other drawing. When you select this option, the layout drawing is saved in a file. The objects in the drawing correspond to the entities in the Builder window. You can now load and edit the drawing like you would any other.

Although you can freely change all the objects in the Builder's drawing, you should exercise caution because drastic changes can prevent the Builder from working properly.

To use the modified layout drawing, set the *GLG_EDITOR_LAYOUT* environment variable to the name of the file you saved.

If the Builder does not work properly and you want to revert to the previous layout, unset the *GLG_EDITOR_LAYOUT* environment variable.

This option is not available in the Basic Edition of the Builder.

Save HMI Layout

Save HMI Layout is similar to *Save Layout*, but is available only in the OEM version of the Builder and saves the drawing that defines the layout and appearance of the of the HMI Configurator.

To use the layout drawing, set the *GLG_HMI_EDITOR_LAYOUT* environment variable to the name of the HMI layout file.

Help

Online Reference

On Windows, *Online Reference* starts a browser with the GLG Online Documentation page. On Linux/Unix, it displays a dialog with information on how to access GLG Documentation.

About GLG Toolkit

About GLG Toolkit displays a dialog with the GLG Toolkit version information.

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

T

Y

Z