The GeoSim Interface Library (GIL)

Programmer’s Manual, Version 2.0

Project GeoSim
David T. Hines  John B. Raley  James M. A. Begole
Colin A. Klipsch  Clifford A. Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061
email: geosim@cs.vt.edu
Contents

1 Introduction 1

2 GIL Basic Structure 3
  2.1 Event Loop .............................................. 3
  2.2 GIL Basic Interface Elements .......................... 3
  2.3 Interface File ......................................... 4

3 GIL Features 6
  3.1 Look and Feel of GIL Buttons ....................... 6
  3.2 Log Files .............................................. 6
  3.3 Backing Store ........................................... 6
  3.4 Redraw Function and Multi-Window Environments .... 7
  3.5 Drag Functions ......................................... 8
  3.6 Window Relative vs. Absolute Coordinates ........... 8
  3.7 Popup Windows ......................................... 9
  3.8 Active vs. Inactive Windows .......................... 9
  3.9 Enabled vs. Disabled vs. Invisible Buttons, Drag Areas, and Menu Items 9
  3.10 Checklist Menus .................................... 9
  3.11 Picklists .............................................. 10
  3.12 Fonts .................................................. 11
  3.13 Bitmap Images ....................................... 12
  3.14 Linking Function Names to Function Pointers ...... 12
  3.15 Generic Database Reader ............................. 12

4 Interface File: An Example 15
  4.1 Color Declarations .................................... 15
    4.1.1 Color Palette Declaration ........................ 15
    4.1.2 Color Alias Declarations ......................... 16
    4.1.3 Color List Declarations .......................... 17
  4.2 Interface Element Declarations ..................... 18
    4.2.1 Menu Declarations ................................ 18
    4.2.2 Window Declarations .............................. 19
    4.2.3 Button Declarations ............................... 22
    4.2.4 Field Declarations ................................. 24
    4.2.5 Drag Area Declarations ............................ 25
    4.2.6 Label Declarations ................................. 25
    4.2.7 Line and Rectangle Declarations ................ 26
    4.2.8 Draw Order of Labels, Lines, and Rectangles .... 28
    4.2.9 Independent Function Declarations .............. 28
    4.2.10 Redraw Function Declaration .................... 29
    4.2.11 Miscellaneous Parameter Declarations ......... 29

5 GIL Image Format 31
  5.1 Runlength Encoder .................................... 32

6 Portability Issues 33
7 GIL Constants and Data Types
  7.1 Constants .................................................. 34
  7.2 Data Types .................................................. 34
    7.2.1 Enumerated Types ....................................... 35
    7.2.2 Handles .................................................. 35
    7.2.3 Character Strings ....................................... 35
    7.2.4 Boolean Values .......................................... 37
    7.2.5 GSColor ................................................ 37
    7.2.6 FormatList ............................................. 38
    7.2.7 GSPicture ............................................... 38
    7.2.8 IntrFunction ........................................... 38
    7.2.9 ControlType ........................................... 39

8 GIL Function Library ......................................... 40
  8.1 Initialization, Start-Up and Shutdown .................... 40
    8.1.1 Interface Initialization ................................. 40
    8.1.2 Event Processing ....................................... 40
    8.1.3 Shutdown .............................................. 41
  8.2 Functions for Manipulating Interface Elements .......... 42
    8.2.1 Functions for Window Support ......................... 42
    8.2.2 Functions for Button Support .......................... 44
    8.2.3 Functions for Menu Support ............................. 46
    8.2.4 Functions for Drag Area Support ....................... 48
    8.2.5 Functions for Field Support ............................ 50
    8.2.6 Scrolling List Functions ............................... 52
    8.2.7 Color List Functions ................................... 55
    8.2.8 Alert and Help Screen Display Utilities ............. 56
  8.3 Functions for Screen Text, Numbers, and Graphics ....... 58
    8.3.1 Text and Numeric Output Functions .................... 58
    8.3.2 Functions for Graphics Support ........................ 60
    8.3.3 Functions for Picture Support ........................ 63
    8.3.4 Keyboard Support Utilities ............................ 65
    8.3.5 Mouse Support Utilities ................................ 66
    8.3.6 Miscellaneous Support .................................. 66
  8.4 Functions for File Access ................................ 70
    8.4.1 Opening Other Files ................................. 70
    8.4.2 Log Files ........................................... 70
    8.4.3 Database Reader Routines ............................. 71
    8.4.4 Opening Other Files ................................. 73

9 giltest ...................................................... 74
  9.1 Interface File for giltest ................................ 74
  9.2 Source Code for giltest .................................. 77
1 Introduction

The GeoSim Interface Library (GIL) is a small, easy to use set of functions for building a graphical user interface (GUI). It was originally designed to support the user interface needs of Project GeoSim, a series of interactive simulations for introductory geography classes. The original design for GIL was driven by the need to develop easy to use software that was portable to a variety of computing environments. It was used by a team of programmers engaged in rapid prototyping and frequent change to their user interfaces. Thus, GIL had to make it easy for the programmer to set up an interface quickly for experimentation, as well as allow easy changes to the positions of interface elements.

Many aspects of GIL’s functionality are limited as compared to full graphical user interface design systems such as Motif, the Macintosh Toolbox or Microsoft Windows API. GIL limits the application developer to a single, rectangular piece of real estate (640 x 480 pixels), a simple color model, and a truly crude font model. However, GIL successfully achieves its primary goals of being portable and easy to use. It currently runs under MS-DOS, the Macintosh Toolbox, and several implementations of X Window. It has been successfully used not only for the entire series of GeoSim simulations, but also as the prototype interface for several research projects and as the graphical user interface package for students in Computer Science programming courses. In addition, the price is right.¹

GIL’s relationship to a particular computing environment is illustrated by Figure 1. An application programmer writes an application using GIL for all user interface interactions. If no platform-dependent functionality has been introduced into the application code, then the program should compile and run without modification under any GIL supported programming environment. Porting GIL itself to a new computing environment requires that a small platform-dependent base library be re-written.

Typographic Conventions. The following typographic conventions are observed in this manual:

- Italics Font is used for formal parameter names, emphasis, and to introduce new terms.
- Teletype Font is used for actual parameter names, file excerpts, file names, and function prototypes.
- Bold Font is used for proper titles, and for emphasis when necessary.

¹GIL is freely available via anonymous FTP, gopher or any World Wide Web client from geosim.cs.vt.edu, URL http://geosim.cs.vt.edu
Figure 1: Abstraction layers for application programs using GIL: from User to Hardware.
2 GIL Basic Structure

2.1 Event Loop

GIL, like most GUI systems, is event driven. The core of GIL is the event handler, or event loop. From GIL's point of view, events are low level, and correspond roughly to inputs from any one of several hardware devices. For purposes of exposition, an event viewed from this perspective will be referred to as a system event. The event loop is responsible for detecting and "handling" all system events. "Handling" a system event involves determining what application program function — if any — to execute in response to the event. This determination is made on the basis of a mapping, or table, declared in the application program. (See Section 3.14 for details about declaring this table.) Note that there may be many system events which do not execute an application program function.

From the application programmer's perspective, an event is generally any manipulation of an interface element by the user, such as a button press, a mouse "drag", a mouse "click", etc. The application programmer's connotation of the word event will be described by the term user event. Note that each user event must cause execution of an application program function. Functions associated with user events are sometimes known as callback functions. Each manipulable interface element is associated with exactly one application program function, which is executed in response to any user event associated with the element. Manipulable GIL interface elements are called control elements. Non-manipulable elements are called static elements.

So, if GIL is responsible for handling events, what is the application programmer's role? Rather broadly, the application programmer is responsible for the following:

1. Declaring all interface elements. This is done via the interface file, which is described in Section 2.3. Interface files can be written by hand with any text editor. Also, interface files can be generated with Graphical Builder a GIL development tool. Graphical Builder allows the developer to "draw" an interface. Graphical Builder then produces an interface file matching the user's drawing. Usually, generating interface files with Graphical Builder is faster and less error-prone than writing them by hand.

2. Writing an application program function for each control element. The GIL functions are available to make this easier, and to make the application code portable among several computing environments. The GIL functions are described in Section 8.

3. Mapping each interface element to its function. This is done via the FuncNames table, which is described in Section 3.14.

2.2 GIL Basic Interface Elements

A graphical user interface is composed of elements. A GUI element is roughly analogous to a discrete physical object in the real world; for that reason, the word "object" is often used in place of "element" in interface descriptions. However, "object" has its own connotations in the world of
computer science, so we prefer to use “element”. GIL offers the following simple types of graphical user interface elements:

Windows are named, rectangular regions of the screen within which other elements may be grouped.

Menus are lists of functions or program attributes from which a user may select.

Buttons are graphical elements to be “pressed” by the mouse. “Pressing” a button causes some function in the application program to be called. Each button belongs to a particular window.

Fields are rectangular regions in which output is displayed.

Drag Areas are rectangular regions of the screen within which a user may perform mouse operations such as pointing, dragging, and clicking.

Labels are static text phrases which are fixed in a particular position within a window, and are drawn automatically by GIL whenever the window which contains them is drawn.

Rectangles and Lines which are declared in the interface file are drawn automatically by GIL whenever the window which contains them is drawn. Rectangles and lines may also be generated on-the-fly by an application program.

2.3 Interface File

GIL is characterized by the use of a separate file for declaring GUI elements. The advantage of using a text file is that changes can be made to the elements without recompiling source code, thus allowing these changes to be made much faster. This text file is referred to as the interface file. The extension .inf is conventionally used to designate an interface file name.

The general syntax of a declaration in an interface file is:

\[ \text{interface-element-variable ( argument-list )} \]

where the arguments in argument-list are separated from each other by commas or whitespace (an arbitrary number of tabs or blank spaces). A declaration may span multiple lines; however, each argument in the declaration can occupy only one line. The interface file parser is case insensitive. Section 4 presents an example of an interface file.

Interface File Argument Types. There are three types of arguments in an interface file argument list:

Integers are sequences of digits, optionally preceded by a unary minus (‘-’). Negative integers are uncommon in interface files.

Names consist of letters, numbers, and the underscore (‘_’). Names must start with a letter.

Strings begin and end with the ‘|’ delimiter. A string may contain any ASCII printing character except ‘|’.
Several names are reserved; they cannot be used in a name argument. Reserved names are:

- ALIAS
- BUTTON
- BUTTONSTUFF
- CANCEL
- COLORLIST
- DRAGAREA
- HIGHLIGHT
- FIELD
- FUNCTION
- LABELS
- LINE
- MSG
- PALETTE
- POPUP
- RECT
- REDRAWFUNC
- WINDOW

Reserved names are case-insensitive. For example, `redrawfunc` and `RedrawFunc` are reserved.
3 GIL Features

3.1 Look and Feel of GIL Buttons

The look and feel of an interface element is how it looks on the screen, and how it responds when a user manipulates it. The buttons in GIL have been designed to resemble push buttons on real mechanical devices. Their edges can be shaded to give the appearance of light falling on a three-dimensional object. The top and left edges are usually given a bright color, to resemble light; the bottom and right edges are usually given a dark color, to resemble shadow. Thus, the buttons appear to be “sticking out” from their windows. The BUTTONSTUFF declaration in the interface file sets the button edge colors for all buttons in the interface.

A button’s shading changes when it is pressed: the edge colors are inverted, so that the button looks as though it is pushed in, or depressed. When the button is released, its appearance reverts back to normal. A depressed button also regains its normal appearance if the mouse cursor is moved away from it while the mouse button is still down (i.e., the button essentially becomes “undepressed”). If the cursor is placed back on the mouse before the mouse button is released, however, the GIL button will again look pushed in.

GIL buttons respond in one of two ways, depending on whether or not they are repeatable. The function associated with a non-repeatable GIL button is executed only when the mouse button is released while the GIL button is depressed. The cursor must still be on the GIL button when the mouse button is released in order for the button’s function to be called.

A repeatable GIL button causes its function to be executed before the mouse button is released. Furthermore, the function will be executed repeatedly until the mouse button is released. The rate at which the function is repeated starts out slow, and gradually increases over a few seconds. The speed-up rate is controlled by the SPEEDUP declaration in the interface file. (The SPEEDUP declaration is described in Section 4.2.11.)

3.2 Log Files

GIL provides two log files for the application: an error log file (error.log) and an output log file (output.log). Both GIL and the application may log errors to the error log. The application alone writes to the output log. error.log is conventionally used for debugging purposes, and GIL's own messages are written there. output.log is available for the application’s needs.

3.3 Backing Store

Backing store is memory used to save a screen area when it is overdrawn by a GIL window or menu. When the window or menu is later removed, the image saved in backing store is restored to the screen. Restoring the screen from backing store is quicker and smoother in appearance than redrawing it from scratch. However, storing screen images requires additional memory, which may be scarce on some systems.
Popup menus automatically use backing store. An application decides at runtime whether to use backing store for its windows.

**GIL** provides the programmer with five backing store areas. That is, up to five windows and/or popup menus at a time may use backing store memory. Each backing store has an integer identifier in the range 1 to 5. Popup menus automatically use level 1. Windows can be assigned to this or other backing stores as desired.

To use backing store to save the portion of the screen overdrawn by a window, an application programmer must do three things:

1. Call `GSinterfaceinit` with `bstore = BSTORE`. This enables backing store for the application. See page 40 for more information on `GSinterfaceinit`.

2. In the window's interface file declaration, set `bstore` equal to the identifier of the backing store that the window will use. See page 4.2.2 regarding interface file declarations for windows. Note that two windows that need to use backing store simultaneously must use different backing stores.

3. When the window is drawn with `GSDrawNamedWindow`, the `bstore` argument must equal `BSTORE`. See 8.2.1 for more information about `GSDrawNamedWindow`.

Care should be taken to avoid backing store “collisions”. A collision is a runtime error that occurs when a window or menu tries to use a backing store level that is already in use. A collision will cause the application to shut down.

### 3.4 Redraw Function and Multi-Window Environments

When a **GIL** application runs in a multi-window environment (such as Macintosh or X Window), **GIL** graphical output is sent to a “native” window, one which is created by the environment's window manager. It is possible for one window to cover the **GIL** application’s window; when the **GIL** window is uncovered, its contents will need to be redrawn.

**GIL** accepts expose messages from the host environment’s window manager. **GIL** responds to these messages by calling the application-defined redraw function. This redraw function should call `GSDrawNamedWindow` for all windows which are currently visible, and then redraw anything that appears in active **GIL** windows but is not an interface element. For example, an application which draws a figure into a window would need to redraw the figure from its redraw function.

Note that in order to reproduce graphical output, an application must preserve information about the output. It is not sufficient simply to draw output and then forget about it. See Section 9 for an example of an application’s redraw function.

To activate a redraw function, an application programmer must register the function in the **FuncNames** array, and declare the function in the application's interface file. See Section 4.2.10 for the syntax of the redraw function declaration.
Declaring a redraw function is optional; if no redraw function is declared, GIL automatically redraws active windows and their interface elements. An application which only displays GIL interface elements does not need a redraw function. Also, a GIL application running under DOS does not need a redraw function, since there are no competing applications that might cover the GIL application. However, it is a good idea to write a redraw function for every GIL application. Including a redraw function makes the application's code more portable.

3.5 Drag Functions

A drag function is associated with a drag area by a declaration in the interface file. A drag function takes the integer parameters \(x\) and \(y\), and the parameter \texttt{status}\textbackslash n, of type \texttt{DragStatusType}. The screen coordinates (relative to the current window) of the mouse cursor are passed in the parameters \(x\) and \(y\) each time the event loop is executed. The type of mouse event which caused the drag function to be called is passed in \texttt{status}. The possible values for \texttt{status} — along with the respective events — are:

- \texttt{DRAG\_INIT} is passed when the user initiates a mouse drag. A mouse drag is initiated by pressing and holding the mouse button down and moving the mouse.
- \texttt{DRAG\_PROCESS} is passed whenever the mouse moves during a drag. A mouse drag continues as long as the mouse button is held down.
- \texttt{DRAG\_FINAL} is passed when the user releases the mouse button after a drag.
- \texttt{MOUSE\_INSIDE} is passed when the mouse cursor moves to a point inside the drag area from a point either inside or outside the drag area while the mouse button is up.
- \texttt{MOUSE\_CLICK} is passed when the user releases the mouse button without having moved the mouse while the button was down.
- \texttt{MOUSE\_OUTSIDE} is passed when the mouse cursor moves to a point outside of the drag area from a point either inside or outside the drag area while the mouse button is up.

3.6 Window Relative vs. Absolute Coordinates

Generally, an application directs output (strings, graphics) to a named window. The location of such output is meant to be relative to the window, so that moving the position of the window will not affect the relative placement of graphics within the window. The GIL drawing functions, therefore, take window relative coordinates as opposed to absolute screen coordinates. However, the GIL event handler sends drag area relative coordinates to a drag area's function, because these events are typically processed relative to the drag area's position. If the application needs to find the window relative coordinates of such an event, it can retrieve the window relative origin of the drag area or button. (See \texttt{GSgetbuttonspecs} and \texttt{GSgetcursorxy} in Section 8.3.6 for details.)

The Current Window. In order to send output to a named window, it must be the current window. Only one window at a time can be current. The current window can be set explicitly by the function \texttt{GSsetCurrentnamedwindow}. Several other GIL functions set the current window implicitly.
3.7 Popup Windows

A popup window is one that "pops" up in front of other windows. Popup windows are typically meant to be on the screen for short periods of time. A GIL popup window is preemp-tive, meaning that other windows (and their buttons and drag areas) are made temporarily inactive until the popup window is removed, usually by some user action. The GIL function \texttt{GSetactiveNamedWindow} can be used to reactivate a window before the popup window is removed.

The interface file declaration of a GIL window says nothing about whether or not it is a popup. Windows only become popups or non-popups when they are drawn. The GIL function \texttt{GDrawNamedWindow} is for drawing windows on the screen; it takes the parameter \texttt{popup}. The value passed to \texttt{popup} determines whether or not a window is drawn as a popup. The legal values for \texttt{GDrawNamedWindow} are \texttt{POPUP} (to draw the window as a popup) and \texttt{NO_POPUP} (to draw the window as a non-popup).

3.8 Active vs. Inactive Windows

A user may only manipulate the controls of active windows. For example, a button may be displayed on screen, but it will not respond to mouse presses unless the window it belongs to is active. The most recently drawn window is considered active by default. Additionally, the application developer may set any number of windows active or inactive as desired with \texttt{GSetactiveNamedWindow} and \texttt{GSetdeactiveNamedWindow}. The application developer is responsible for guarding against overlapping active windows or related problems.

3.9 Enabled vs. Disabled vs. Invisible Buttons, Drag Areas, and Menu Items

Buttons, drag areas, and menu items may be enabled or disabled. In its disabled state, the button label or menu item is \textit{dimmed} (i.e., it uses the disabled text color). The user can see a disabled button or menu item, but the color of disabled text should make its inaccessibility clear. Disabled buttons and menu items may also be made invisible, if the application programmer does not want them to be seen at all.

3.10 Checklist Menus

If a menu is declared in the interface file to be a checklist, then GIL will associate a check mark with each item, which is used as a bool indicator. Selecting an item with an invisible check mark makes the check mark visible, and vice versa. Other than this, GIL treats a checklist menu just like any other menu: each item on any menu is associated with an application program function, which is executed when the item is selected.
Semantics of Checklist Menus. It is up to the application programmer to establish the semantics of any menu. The semantics of a menu are implemented in the application program itself, via the functions associated with the menu’s items. Checklist menus are typically used to set application program parameters which will affect the subsequent behavior of the program. Some checklists represent a list of bool parameters. In this case, the check mark for each parameter is generally visible when the parameter is true, or on, and invisible when the parameter is false, or off. Other checklists represent a list of legal values for a single parameter. In this case, only one value can have a check mark next to it at a time (the application programmer must remove old check marks when a new value is selected.)

The application giltest is a simple example of GIL. giltest has both types of checklist menus mentioned above. In the “box style” menu, the options “Draw Black Border” and “Fill With Color” are independent of each other; both can be checked or unchecked at the same time, or exactly one can be checked.

The “fill color” menu in giltest allows the user to select a color. Exactly one color may be checked at a single time. Note that giltest is responsible for unchecking colors when a different color was selected.

3.11 Picklists

An application which needs to display a large number of items (more than will fit in a window), or a number of items which can change, can use a picklist to display the items. Picklists are a common interface component in many graphical user interfaces.

Picklists can contain any number of items. The picklist displays as many items as will fit in its display area. Also, the picklist tracks mouse activity inside its display area, reporting which item is currently under the mouse, and reporting when the mouse is clicked on an item.

A picklist is almost always used in conjunction with a scroll bar and scroll buttons. A scroll bar moves the picklist display proportionally through the picklist’s items. Scroll buttons move the picklist display either “down” or “up” by one item, each time the button’s action executes. Together, the picklist/scroll bar/scroll button system is called a scrolling list.

In GIL, a scrolling list is a composite interface element. Composite interface elements are not declared in a GIL interface file. They are made of several basic GIL interface elements. A scrolling list requires these GIL interface elements: two drag areas, one for the picklist and one for the scroll bar; and two buttons, one for scrolling up and one for scrolling down.

GIL handles most scrolling list tasks, including drawing and responding to user scrolling. GIL references list components through handles. A handle is simply a unique identifier given to each list component. GIL creates handles every time a scrolling list is created. Most GIL scrolling list functions require a handle as a parameter. Specifically, GIL uses two types of list handles: GSlistHandle and GSscrollHandle.

In order to support scrolling lists, the application programmer must do the following:
1. Declare in the application’s interface file a drag area for the picklist, another drag area for the scroll bar, and two buttons for scrolling.

2. As soon as the window containing the scrolling list is drawn, call GScclistsetup with appropriate parameters. In particular, store the GSlistHandle and GScclistHandle returned from GScclistsetup.

3. In the picklist drag area’s function, call GShandlelist, passing the GSlistHandle returned from GScclistsetup as a parameter. Call GShandlesclist in the scroll bar drag area’s function, with the GScclistHandle returned from GScclistsetup as a parameter. In the up and down buttons’ functions, call GShandleupbutton and GShandedownbutton, respectively, passing the GScclistHandle as a parameter.

4. In the application’s redraw function, call GScdrawsclist for every visible scrolling list. As the parameter to GScdrawsclist, pass the GScclistHandle returned from GScclistsetup.

5. When the window containing the scrolling list is removed, call GSdissmantlesclist, with the scrolling list’s handle as the parameter.

The GIL functions mentioned above are documented in Section 8.2.6.

There is only one restriction on the drag areas and buttons used for a scrolling list: they all must be in the same window. However, a scrolling list looks better if its scroll bar is immediately to the right of its picklist, with the up and down scroll buttons immediately above and below the scroll bar. Using the | \^ | and | \v | arrow symbols for the text of the up and down arrow buttons, respectively, gives these buttons an appearance which suggests their purpose. See Section 4.2.3 for more information about button arrow symbols.

Among GScclistsetup’s parameters are five function pointers. These functions are supplied by the application programmer to implement various aspects of the scrolling list’s features. They are:

- `label_func` - returns a pointer to the text of a list item.
- `settext_func` - sets the text attributes (font and color) for a list item.
- `in_func` - called when the mouse is over a list item. The item number is passed as a parameter to this function.
- `out_func` - called when the mouse moves out of the picklist area.
- `click_func` - called when the mouse is clicked on a list item. The number of the selected item is passed as a parameter to this function.

### 3.12 Fonts

GIL provides two simple fonts: SMALL and LARGE. LARGE is a boldface version of SMALL, a sans serif typeface. The exact typeface depends on the computer platform on which the program is compiled.
3.13 Bitmap Images

**GIL** uses its own bitmap image file format for raster graphics. This file format is based on Run Length Encoding (RLE) to keep images small. Several pixel resolutions are supported to provide optimal compression and flexibility. The image file format is discussed in Section 5.

**GIL** allows an image to be drawn either directly from a file, or from a **GSPicture** structure in memory. Drawing a picture stored in a **GSPicture** is faster than drawing one stored in a file, but uses more memory. See Section 8.3.3 for documentation on **GIL** picture support routines.

3.14 Linking Function Names to Function Pointers

A GUI element such as a button, drag area or menu item is associated with an application program function that performs some action when the element is manipulated by the user. The function name is specified when the element is declared in the interface file. However, there must be some mechanism for attaching the name of the function as specified in the interface file to a function pointer (address) in the program. This is done using an array of structures — called **FuncNames** — initialized within the application program source code. Each element of the array maps a name to its associated function pointer. Note that independent functions, which are not associated with a particular interface element, (see Section 4.2.9) must also be included in the **FuncNames** array. This approach requires recompilation if either the logical or physical name of the function changes, as well as some coordination between the interface file and the source code. Fortunately, these names rarely change in practice. Figure 2 shows the **FuncNames** array for **gillest**, a simple example of a **GIL** application.

Initialization of **NumFuncs**. The application programmer must declare and initialize a global integer variable named **NumFuncs**. **NumFuncs** tells **GIL** the number of functions in the **FuncNames** array. **GIL** supplies a macro **GSnumifuncs** to make initialization of **NumFuncs** simple and convenient for the application programmer. Initialization of **NumFuncs** via **GSnumifuncs** is shown at the bottom of Figure 2.

3.15 Generic Database Reader

**GIL** was originally developed to support **Project GeoSim**, whose simulations typically require extensive databases. Thus, various database support functions were developed early on. The design philosophy was that data files should be human as well as machine readable. Therefore, the **GIL** database support functions use an ASCII-based data formatting scheme. It quickly became the convention to use this same file structure and these same support utilities for other **GIL** application needs such as application configuration files. Figure 3 shows a database file for the **Project GeoSim** application **Mental Maps**. Section 8.4.3 describes the database reader functions.
IntrFunction FuncNames[] = {
    /* name in .inf file   function pointer */
    { "again_func", do_again },
    { "do_nothing", do_nothing },
    { "intro", do_intro },
    { "color_menu", show_colors },
    { "clear_draw_func", clear_draw },
    { "clear_text_func", clear_text },
    { "quit_prog_func", do_quit },
    { "done_func", alert_done },
    { "box_menu", show_box_styles },
    { "funcs_menu", show_funcs_menu },
    { "kbd_hdlr", (bool(*)(()))do_text },
    { "draw_func", (bool(*)(()))do_draw },
    { "set_color_func", set_color },
    { "toggle_fill_func", toggle_fill },
    { "toggle_border_func", toggle_border }
};

int NumFuncs = GSnumfuncs(FuncNames);

Figure 2: The FuncNames array for giltest.
<table>
<thead>
<tr>
<th>#keyword</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>database=</td>
<td>MM_cdata # ID tag for database; passed to GStgenreader</td>
</tr>
<tr>
<td>mapscale=</td>
<td>1 # map scale factor (x’s, y’s multiplied by this)</td>
</tr>
<tr>
<td>mapoffx=</td>
<td>1 # map horizontal offset (added to x’s)</td>
</tr>
<tr>
<td>mapoffy=</td>
<td>15 # map vertical offset (added to y’s)</td>
</tr>
<tr>
<td>numcities=</td>
<td>24 # number of city records to follow</td>
</tr>
<tr>
<td>format=</td>
<td># signals that format list follows</td>
</tr>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>225</td>
<td>207</td>
</tr>
<tr>
<td>399</td>
<td>177</td>
</tr>
<tr>
<td>280</td>
<td>107</td>
</tr>
<tr>
<td>302</td>
<td>290</td>
</tr>
<tr>
<td>279</td>
<td>248</td>
</tr>
<tr>
<td>269</td>
<td>292</td>
</tr>
<tr>
<td>237</td>
<td>217</td>
</tr>
<tr>
<td>312</td>
<td>132</td>
</tr>
<tr>
<td>321</td>
<td>127</td>
</tr>
<tr>
<td>301</td>
<td>186</td>
</tr>
<tr>
<td>260</td>
<td>169</td>
</tr>
<tr>
<td>367</td>
<td>150</td>
</tr>
<tr>
<td>275</td>
<td>326</td>
</tr>
<tr>
<td>283</td>
<td>163</td>
</tr>
<tr>
<td>230</td>
<td>172</td>
</tr>
<tr>
<td>289</td>
<td>127</td>
</tr>
<tr>
<td>331</td>
<td>151</td>
</tr>
<tr>
<td>293</td>
<td>141</td>
</tr>
<tr>
<td>263</td>
<td>65</td>
</tr>
<tr>
<td>236</td>
<td>134</td>
</tr>
<tr>
<td>305</td>
<td>110</td>
</tr>
<tr>
<td>206</td>
<td>168</td>
</tr>
<tr>
<td>279</td>
<td>74</td>
</tr>
<tr>
<td>281</td>
<td>112</td>
</tr>
</tbody>
</table>

Figure 3: The India city data file for Mental Maps.
4 Interface File: An Example

The easiest way to introduce the parts of an interface file is by means of a simple example. The excerpts used here are from an interface file for a simple program called *giltest*. The purpose of *giltest* is to provide exposition of the basic types of interface elements available with GIL.

**Interface Files and the Graphical Builder.** All GIL developers should have some familiarity with GIL interface files. However, we strongly encourage developers to use the Graphical Builder to create interface files, instead of hand-coding them. With Graphical Builder a developer can generate an interface quickly, and can study its “look and feel” while creating it. Also, the interface files created with Graphical Builder are free from syntactical errors which inevitably occur in hand-coded interface files.

**Comments in the Interface File.** Comments in interface files begin with a pound sign (#). All text to the right of the pound sign is ignored. Comments can begin in any column.

4.1 Color Declarations

4.1.1 Color Palette Declaration

An interface file typically begins with the declaration of a color palette. The color palette defines the specific colors available for use in the interface. A palette consists of up to 256 colors. The number of colors in the palette declaration should correspond to the number of colors specified in the GIL initialization. (See Section 8.1.1). Each color in the palette is specified by a set of Red-Green-Blue (RGB) values. Each RGB value must be in the range 0-63. A color declaration has the following syntax:

\[
(\text{color-name} \quad \text{red-value} \quad \text{green-value} \quad \text{blue-value})
\]

All of the parameters are mandatory. The parameters are:

- **color-name**: name - the name of the color.
- **red-value**, **green-value**, **blue-value**: integer - the red, green, and blue components of the color.

The palette declaration must begin with the word PALETTE. The color declarations follow, one color per line. The word END must appear after the last color declaration. The color palette declaration for *giltest* appears in figure 4.
4. INTERFACE FILE: AN EXAMPLE

PALETTE
# R G B
#
(black 0 0 0)
(dgray 50 45 45)
(mgray 30 30 55)
(lgray 53 53 53)
(aqua 0 50 50)
(lblue 10 10 55)
(peach 63 47 50)
(blue 0 0 36)
(green 0 36 0)
(lpurple 63 50 63)
(red 55 0 0)
(yellow 63 63 21)
(lcyan 50 59 63)
(magenta 44 6 44)
(sandy 63 51 51)
(white 63 63 63)
END #PALETTE

Figure 4: The 16-color palette for giltest.

4.1.2 Color Alias Declarations

Note that the color names used in the palette declaration for giltest are general in nature. These names attempt to describe the appearance of the colors, not the ways in which these colors are used in the application. It is better to indicate the use of a color with an aliased name. Use of color aliases in the interface file makes it easier to modify the colors of interface elements, just as using named constants makes your code easier to change. Figure 5 shows the color alias declarations for giltest.

For an example of how aliases can make it easier to modify an interface, note that altering the alias ALIAS (bbg lblue) to ALIAS (bbg yellow) would change the background color of all buttons in the interface from blue to yellow. Likewise, the appearance of menus, windows, etc. can be easily and consistently modified by changing the appropriate alias declarations. The power inherent in color aliases is especially helpful in applications with large interfaces.

The syntax for a color alias declaration is as follows:

ALIAS ( color-name alias-name )

All of the parameters in color alias declarations are mandatory. The parameters for color alias declarations are:

color-name: name - the name of the color referenced by alias-name.
# alias        color
#            ------        ------

# Window colors.
ALIAS (rootbg  aqua   )  # background color for root window
ALIAS (rootfg  black  )  # foreground color for root window
ALIAS (wbdr    black   )  # border color for windows

# Button colors.
ALIAS (bbg     lblue   )  # background color for buttons
ALIAS (befg    white   )  # text color for enabled buttons
ALIAS (bdfg    mgray   )  # text color for disabled buttons

# Menu colors.
ALIAS (popefg  black   )  # text color for enabled popup menu items
ALIAS (popdfl  dgray    )  # text color for disabled popup menu items
ALIAS (popbg   peach   )  # background color for popup menus
ALIAS (popbdr  black   )  # border color for popup menus

# Color List colors.
ALIAS (fillcolor0 black  )  # first color in menu
ALIAS (fillcolor1 yellow)  # second color in menu
ALIAS (fillcolor2 white  )  # third color in menu
ALIAS (fillcolor3 lblue   )  # fourth color in menu
ALIAS (fillcolor4 red    )  # fifth color in menu

# Other colors.
ALIAS (text    black   )  # color of text in text entry field
ALIAS (dragbg  white   )  # background color of drag drag area.

Figure 5: Color Aliases for giltest.

alias-name : name - the alias which references color-name.

4.1.3 Color List Declarations

Color lists are used in GIL to establish logical orderings of color. These orderings are typically independent of the order of the colors in the palette declaration. Uses for color lists include graph coloring and support for color selection menus. Giltest uses a color list to support a menu of fill colors. The format for a color list declaration is as follows:

COLORLIST ( color0 color1 color2 color3 color4 )

All of the parameters in a color list declaration are mandatory. The parameters for color list declarations are:

color0, color1, color2, color3, color4 : name - five color names (either palette names or aliases).
There is one color list declared in the interface file for *giltest*:

```
COLORLIST (fillcolor0 fillcolor1 fillcolor2 fillcolor3 fillcolor4)
```

### 4.2 Interface Element Declarations

Once the colors are set, interface elements can be declared. Note that the element declarations all contain either color palette names or their aliases. This means that the color declarations *must* precede the element declarations in an interface file.

#### 4.2.1 Menu Declarations

Menus are lists of functions from which a user may select. Menu items can be disabled or made invisible. Check-list menus, which allow a user to toggle program parameters, are also available. A menu is declared as a **POPUP** in the interface file. Menus do not belong to any particular window. A menu declaration has the following syntax:

```
POPUP ( name x y efgcolor dfgcolor bgcolor bdrcolor checklist? [func] )
```

Except for *func*, all parameters in menu declarations are mandatory. The menu declaration parameters are:

- `name`: *name* - the name of the menu.
- `x`: *integer* - location of the left edge of the menu.
- `y`: *integer* - location of the top of the menu.
- `efgcolor`: *name* - the text (foreground) color for enabled menu items.
- `dfgcolor`: *name* - the text (foreground) color for disabled menu items.
- `bgcolor`: *name* - the background color of the menu.
- `bdrcolor`: *name* - the color of the menu’s border.
- `checklist?`: *integer* - 1 if the menu is a checklist; 0 otherwise.
- `func`: *name* - an optional function for the event handler to call whenever the menu is closed.

Each item of a menu is declared as a **PLABEL** (*Popup LABEL*). A **PLABEL** declaration is associated with the nearest **POPUP** declaration above it in the interface file. The syntax for a **PLABEL** declaration is as follows:

```
PLABEL ( name enable? func text )
```

All parameters in a **PLABEL** declaration are mandatory. The **PLABEL** parameters are:
name : name - the name of the menu item. The item name is passed as a parameter to
func, the item’s function.

enable : integer - 1 if the menu item should be enabled at program startup; 0 otherwise.

func : name - the function which is executed when this menu item is selected.

text : string - the label text to show for this menu item. Label text may contain spaces.
    The label text must be delimited by the ‘|’ character. All characters between the ‘|’
    characters will be part of the label text.

There are three popup menus used in giltest. One of them is a regular menu (as opposed to a
checklist menu):

POPUP (funcs_menu 9 43 popefg popdfg popbg popbdr 0)
    LABEL (ClearText 0 clear_text_func |Clear Text Field|)
    LABEL (ClearDraw 0 clear_draw_func |Clear Draw Area|)
    LABEL (QuitProg 1 quit_prog_func |Quit|)

These declarations declare the Functions menu, which contains three items, and pops up at
position (9, 43) on the screen when the button labeled Functions is pressed. The items are la-
beled Clear Text Field, Clear Draw Area, and Quit. When selected, these items cause the
application program functions linked with the names clear_text, clear_draw, and doquit to be
executed, respectively. (See Section 3.14 concerning function name mapping.) The color charac-
teristics of the menu are defined by the color aliases popefg, popdfg, popbg, and popbdr.

4.2.2 Window Declarations

Windows are named, rectangular regions of the screen within which other elements may be grouped.
The syntax of a window declaration is as follows:

WINDOW ( name x y wd ht fgcolor bgcolor bdrcolor bdrsize active? bstore )

All of the parameters in a window declaration are mandatory. The parameters for window decla-

name : name - the name of the window.

x : integer - location of the left edge of the window.

y : integer - location of the top of the window.

wd : integer - the width — in pixels — of the window.

ht : integer - the height — in pixels — of the window.

fgcolor : name - the text color of the window. This is used only for special help windows
    (see Section 4.2.2) and the login window (see Section 4.2.2).

bgcolor : name - the background color of the window.
bdrcolor: name - the color of the window’s border.
bdsize: integer - the width – in pixels – of the window’s border.
active?: integer - 1 if the window should be active at program startup; 0 otherwise. The buttons, drag areas, etc. in a window can only be manipulated when the window is active.
bstore: integer - the (integer) identifier of the backing store that the window will use to save the portion of the screen that it covers when it is drawn (see Section 3.3). 1, 2, 3, 4, and 5 are the legal backing store identifiers. Use 0 if the window will not use backing store. 0 should be used for windows that will never be removed from the screen during program execution. Note that this parameter only determines which backing store can be used when the window is drawn, not whether it will be used.

Note that while the parameter fgcolor is not used on most windows, it must have a corresponding argument in every window declaration. For windows which do not use fgcolor, any legal color name or alias will suffice as an argument.

The giltest interface contains four windows. Below is an example of window declaration. These windows in turn contain other types of elements, which will be described below. For now, it is only important to note that the declarations of these other elements appear grouped together below the declaration of the window which contains them.

WINDOW (root 0 0 640 480 white rootbg wbdr 2 1 0)
  BUTTON (funcs |Functions| 9 23 105 25 befg bdfg bbg @funcs_menu 1 1 0 C)
  BUTTON (box |Box Style| 9 98 105 25 befg bdfg bbg @box_menu 1 1 0 C)
  BUTTON (color |Fill Color| 9 173 105 25 befg bdfg bbg @col_menu 1 1 0 L)
  LABEL (235 15 white |GeoSim Interface Library (GIL) Test| black)
  LABEL (550 15 rootfg |GeoSim|)
  FIELD (version 450 5 60 19 rootfg)
  FIELD (fillcolor 99 178 10 10 rootfg)

The declaration above is for the root window, which defines the screen area of a GIL application. Its text color argument is white, although this parameter is not used for this window. Its background color is defined by the color alias rootbg, which is associated with the color aqua by an earlier ALIAS declaration. It has a two-pixel wide border, the color of which is defined by the alias wbdr. It is active at the start of the program. It cannot use backing store, since its bstore argument is 0. (It does not need backing store because it is never removed from the screen.) It contains three buttons, two labels, and two fields.

Root Window  Every application that uses GIL must have a window named root. It must be the first window declared in the interface file. The location of the left edge and top of the root window must be 0. The width and height of the root window define the available screen real estate for a GIL application. Currently, the root window must be 640 pixels wide and 480 pixels high.
Help Messages and Help Windows  GIL uses specific window names for its help, message, and alert functions. The window named help is used by the function GSShHelpButton, which loads help text from a file. The text can be of arbitrary length; GSShHelpButton will break it into pages.

The message window is used by GSShHelpButton to display a short message (up to three lines). The message window should remain visible anytime a help message can be displayed. GSShHelpButton also calls GSShHelpButton, so it also needs the message window.

Messages for the message window are declared in the interface file. Each declaration consists of three lines of text; if the message has fewer than three lines, make the declaration as shown below, but leave the extra lines blank.

The syntax of a message declaration is as follows:

\begin{verbatim}
MSG ( hmx0  messagetextline1 )
MSG ( hmx1  messagetextline2 )
MSG ( hmx2  messagetextline3 )
\end{verbatim}

The \texttt{hmx} in \texttt{hmX0}, \texttt{hmX1}, and \texttt{hmX2} is the number of the message. This number can be any unique positive integer; it is used as the argument to GSShHelpButton.

All parameters in message declarations are mandatory. The message declaration parameters are:

\texttt{hmX}: \texttt{name} - the message identifier.

\texttt{messagetextlineX}: \texttt{string} - the message text line. Message text may contain spaces. The label text must be delimited by the \texttt{"|"} character. All characters between the \texttt{"|"} characters will be part of the label text.

Here is an example of a message from \texttt{giltest}:

\begin{verbatim}
MSG (hm10 |Look here for help messages as you go through the program.|)
MSG (hm11 |Press "Again", or type a message, or press "Box Style" to||)
MSG (hm12 |set parameters for drawing boxes in the Drawing Area.|)
\end{verbatim}

The alert window is used by GSShAlert to display warning or error messages. This window is normally out of sight; it pops up only when a message needs to be displayed, and disappears again when the user acknowledges the message. An alert message declaration is similar to the declarations shown above; however, they contain only one line, and may be given any legal identifier name (See Section 2.3 regarding legal identifier names):

\begin{verbatim}
MSG ( messagename |message|text|)
\end{verbatim}

\texttt{giltest} has one alert message, which is used to signal a user that he or she has exceeded the maximum number of characters in an input string:
MSG (typedtoofar |You have reached the end of the text box.|)

Note that the application programmer must declare these windows in the interface file in order to use their corresponding functions. GIL knows the names of the windows, but does not automatically create them.

**Login Window** Some application programmers may want users to “log in” to an application. GIL provides a skeletal mechanism for allowing a user to log in. The GIL function Gsdologinscreen initiates this mechanism. GIL requires the application programmer to declare a window named login in the interface file for the login mechanism to work. The application programmer must also declare a field named login; this field is used to echo user keyboard input.

### 4.2.3 Button Declarations

Buttons are graphical elements to be “pressed” using the mouse. “Pressing” a button causes some function in the application program to be called. Each button belongs to a particular window. When GIL reads a button declaration in the interface file, it associates the button with the most recently declared window. A button can be disabled — in which case it is visible, but cannot be pressed — or made invisible. When a button is disabled, its label is printed using dfgcolor instead of efgcolor.

The syntax for a button declaration is as follows:

```
BUTTON ( name |lab| x y wd ht efgycol dfgycol bgcol func bdr? enab? repeat? labpos )
```

All of the parameters for a button declaration are mandatory. The parameters are:

- **name** : name - the name of the button.
- **lab** : string - the label to show on the button. The label text may contain spaces. The label text must be delimited by the ‘|’ character. The label can contain multiple lines. Use the ‘@’ character to signal the start of a new line (e.g., |line1@line2|). An arrow can be placed on a button by including the appropriate escape sequence in the label argument. See below for more details.
- **x** : integer - location of the left edge of the button, relative to the left edge of the window which contains it.
- **y** : integer - location of the top of the button, relative to the top of the window which contains it.
- **wd** : integer - the width — in pixels — of the button.
- **ht** : integer - the height — in pixels — of the button.
- **efgycol** : name - the color of the button’s label when the button is enabled.
- **dfgycol** : name - the color of the button’s label when the button is disabled.
bycol : name - the background color of the button.
func : name - the function which the button causes to execute. If preceded by the @ character, the popup menu func is displayed when the button is pressed.
bdr? : integer - 1 if the button should have a (black) border; 0 if the button should have no border.
enab? : integer - 1 if the button should be enabled at program startup; otherwise, 0. See Section 3.9 regarding enabled vs. disabled buttons.
repeat? : integer - 1 if the function associated with the button should be repeated when the button is held down; otherwise, 0. See Section 3.1 about repeatable buttons.
labpos : name - L to position the first character of the label near the left edge of the button; C to center the label between the left and right edges of the button; R to position the last character of the label near the right edge of the button.

There are five buttons in the giltest interface. The io window contains the following button declaration:

BUTTON (again |Hello Again| 220 75 80 25 bfg bdfg bbg again_func 1 1 0 C)

The name of this button is again; this is the name used within the application program to refer to the button. The label on this button — which tells the application user what the button does — is Hello Again. The upper left corner of the button is located at pixel position (220, 75); this position is relative to the upper left hand corner of the io window. The color characteristics of this button are defined by a series of color aliases. This button causes the application program function linked to the name again_func to be executed. It has a one-pixel-wide border, is enabled at program startup, and is not repeatable.

The root window contains three buttons, including the functions button, declared as follows:

BUTTON (funcs |Functions| 9 23 105 25 bfg bdfg bbg @funcs_menu 1 1 0 C)

The main difference between this declaration and the one for again is the ‘@’ symbol before the name funcs_menu. This at sign indicates that the button is associated with the popup menu named funcs_menu. This means that when the funcs buttons is pressed, the application program function linked to the name funcs_menu is called, and the funcs_menu menu is displayed. Note that the name of the menu must match the name of the function as declared in the interface file.

The other two button declarations in the root window are also for popup menus, and are similar to the funcs button.

Small Font Buttons. The BUTTON declaration is used for buttons which have labels written in the LARGE font. Alternatively, you can use small font buttons. Small font buttons use the SMALL font. To declare a small font button, use BUTTONS in place of BUTTON in your declaration.
Arrows on Buttons  You may want to put an arrow symbol on a button in addition to — or in place of — text. This can be done by using an escape sequence in the button label argument. The escape sequences for arrow symbols are given in the following table:

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Arrow Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Up</td>
</tr>
<tr>
<td>\v</td>
<td>Down</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left</td>
</tr>
<tr>
<td>&gt;</td>
<td>Right</td>
</tr>
</tbody>
</table>

4.2.4 Field Declarations

Fields are rectangular regions. Fields typically specify a position where output is displayed. They allow the location of graphical elements (such as variable labels or pictures) to be defined in the interface file rather than hard-coding these locations within the program. A field does not belong to a particular window; however, the coordinates of a field are relative to the current window at run-time. In this way, the same field definition may be used to locate output within more than one window.

The syntax for a field declaration is as follows:

```
FIELD ( name x y wd hl color )
```

All of the parameters for a field declaration are mandatory. The parameters are:

- `name`: *name* - the name of the field.
- `x`: *integer* - location of the left edge of the field, relative to the left edge of the current window at run time.
- `y`: *integer* - location of the top of the field, relative to the top of the current window at run time.
- `wd`: *integer* - the width — in pixels — of the field.
- `hl`: *integer* - the height — in pixels — of the field.
- `color`: *name* - a color associated with the field. A field is defined by its location and dimensions; it has no color per se. The color parameter can be used at run time to set the color of output (text, lines, solid fill, etc.) which appears in the field.

There are eleven fields in the *giltest* interface. The field *textfg* in the *io* window defines the color and location for echoing a user’s input string, *textbg* defines a background fill area. The important thing to notice is that the syntax for these two declarations is the same; the semantic characteristics of a field are determined entirely by the application code.

```
FIELD (textbg 220 18 126 16 dgray)
FIELD (textfg 222 30 120 370 text)
```
4.2 Interface Element Declarations

4.2.5 Drag Area Declarations

Drag areas are rectangular regions of the screen within which a user may perform mouse operations such as pointing, dragging, and clicking. A user performs a mouse drag by moving the mouse while holding the mouse button down. Each declared drag area is associated with a drag function in the application program (see Section 3.5). A drag area belongs to the most recently declared window.

A drag area declaration has the following syntax:

\[
\text{DRAGAREA ( name } x \ y \ wd \ ht \ bgcolor \ func \ enable? \) }
\]

All of the parameters for a drag area declaration are mandatory. The parameters for are:

- \(name\) : The name of the drag area.
- \(x\) : Location of the left edge of the drag area, relative to the left edge of the window which contains it.
- \(y\) : Location of the top of the drag area, relative to the top of the window which contains it.
- \(wd\) : The width — in pixels — of the drag area.
- \(ht\) : The height — in pixels — of the drag area.
- \(bgcolor\) : The background color of the drag area.
- \(func\) : The function which the drag area causes to execute.
- \(enable?\) : 1 if and only if the drag area should be enabled at program start-up; 0 otherwise.

gi12est has one drag area, which is part of the io window:

\[
\text{DRAGAREA (drawarea 2 191 376 177 dragbg draw_func 1)}
\]

This drag area is named drawarea, and its upper left corner lies at \((2, 191)\) in the io window. It is 376 pixels wide and 177 pixels high. Its background color is defined by the alias dragbg. It causes the application program function linked with the name draw_func to be called when it is manipulated. (See Section 3.5 regarding drag area manipulations.) It is enabled at program start-up.

4.2.6 Label Declarations

Labels are static text phrases which are fixed in a particular position within a window. They are useful for presenting text which should appear every time the window is drawn. Examples are the
name or version number of the program, the name of the individual or organization responsible for creating the program, the title of a graph, etc. The syntax of a label declaration is as follows:

```
LABEL  ( x  y  color  text  [shadowcolor])
```

All of the parameters in label declarations except shadowcolor are mandatory. The parameters are:

- `x`: integer - location of the left edge of the first letter in the label, relative to the left edge of the window which contains it.
- `y`: integer - location of the bottom of first letter of the label, relative to the top of the window which contains it.
- `color`: name - the text color of the label.
- `text`: string - the text of the label. Label text may contain spaces. The label text must be delimited by the '|' character.
- `shadowcolor`: name - the color of the label's shadow. Shadow text appears one pixel below and to the right of the label. No shadow is displayed if this argument is omitted.

There are three labels in the root window. The first is given by the declaration:

```
LABEL (235 15 white |GIL Application Test| black)
```

This label reads “GIL Application Test”, and appears at position (235, 15) in the root window. It is white with a black shadow.

The second label declaration in the root window reads:

```
LABEL (550 15 rootfg |GeoSim|)
```

This label reads “GeoSim”, and appears at position (550, 15) in the root window. It has no shadow, since there is no argument provided for shadowcolor. There is also an unshadowed label in the io window.

**Small Labels** As with buttons, LARGE is the default font for labels. To use the SMALL font for a label, replace LABEL with LABELS.

### 4.2.7 Line and Rectangle Declarations

Lines and rectangles which are declared in the interface file are drawn automatically by GIL whenever the window which contains them is drawn. Thus, rectangles and lines which are permanent
fixtures in a window can be conveniently managed by declaring them in the interface file. Rect-
angles and lines which appear in a window temporarily are best managed using the GIL drawing
functions described in Section 8.3.2. The syntax for a line declaration is:

\[
\text{LINE ( } x0 \ y0 \ x1 \ y1 \ \text{width} \ \text{color} \ )
\]

All of the parameters in line declarations are mandatory. The parameters are:

\(x0\) : integer - the x-coordinate of endpoint 1 of the line, relative to the left edge of the window
which contains it. (Arbitrarily pick one endpoint to be endpoint 1.)

\(y0\) : integer - the y-coordinate of endpoint 1 of the line, relative to the top of the window which
contains it.

\(x1\) : integer - the x-coordinate of endpoint 2 of the line, relative to the left edge of the window
which contains it.

\(y1\) : integer - the y-coordinate of endpoint 2 of the line, relative to the top of the window which
contains it.

\(\text{width}\) : integer - the width — in pixels — of the line. Legal values are 1 and 2.

\(\text{color}\) : name - the color of the line.

There are ten lines declared in the interface file for giltest. We show two of these declarations here
as examples. There is a two-pixel wide horizontal line which runs between x-coordinates 0 and 380
at y-coordinate 368 in the io window:

\[
\text{LINE ( } 0 \ 368 \ 380 \ 368 \ 2 \ \text{black} \ )
\]

There is also a one-pixel wide vertical line which runs between y-coordinates 35 and 16 at x-
coordinate 347 in the io window:

\[
\text{LINE ( } 347 \ 35 \ 347 \ 16 \ 1 \ \text{white} \ )
\]

The syntax for a rectangle declaration is as follows:

\[
\text{RECT ( } x \ y \ wd \ ht \ \text{color} \ \text{bdrsize} \ \text{style} \ )
\]

All of the parameters for rectangle declarations are mandatory. The parameters are:

\(x\) : integer - location of the left edge of the rectangle, relative to the left edge of the window
which contains it.
y : integer - location of the top of the rectangle, relative to the top of the window which contains it.

wd : integer - the width — in pixels — of the rectangle.

ht : integer - the height — in pixels — of the rectangle.

color : name - the color of the rectangle’s border.

bdrsize : integer - the width — in pixels — of the rectangle’s border. Legal values are 1 and 2. Not used for solid rectangles (but argument nonetheless required.)

style : name - “fill” for a solid rectangle, “hollow” for hollow frame rectangle.

There are three rectangles declared in the interface file for giltest. We show two examples here. The first is a solid rectangle; the second is a hollow frame rectangle with a thick border:

```
RECT (218 16 130 20 dgray 2 fill)
RECT (218 16 130 20 wbdr 2 hollow)
```

### 4.2.8 Draw Order of Labels, Lines, and Rectangles

A window’s labels, lines, and rectangles are all static interface elements. This means that they are not changed by the application program code, and that they always are drawn automatically when the window is drawn. They are drawn in the order in which they are declared in the interface file. If two static elements occupy the same screen space, the element which is declared later in the interface file will be drawn “on top”.

### 4.2.9 Independent Function Declarations

These are functions that are not part of any specific interface element. The application programmer may declare an independent function to be called each time GIL iterates its event loop. Independent functions may be activated at program startup with the active? parameter in the interface file. They can also be activated (with GActivateFunction) and deactivated (with GDeactivateFunction) by the program as needed. Note that independent functions must also be included in the FuncNames array. The purpose of the FUNCTION declaration is to make the function eligible to be called by the event loop. This allows an application to provide functionality not tied to a user action, such as an animation or time-based simulation. The syntax of a function declaration is as follows:

```
FUNCTION ( funcname dtime active? [mouseoff?] )
```

The first three parameters are mandatory; the last parameter is optional. The parameters for function declarations are:
4.2 Interface Element Declarations

funcname : name - the name string of the function as declared in the FuncNames array.
dtime : integer - how often to execute the function. Not implemented.
active? : integer - 1 if the function should be active at program startup; 0 otherwise.
mouseoff? : integer - to hide the mouse cursor during execution of the function, use 1, or omit
the argument. If the cursor should remain visible, use 0.

There are two independent functions declared in the giltest interface file:

    FUNCTION (intro 0 1)
    FUNCTION (kbd_hndlr 0 0)

4.2.10 Redraw Function Declaration

Section 3.4 described the need for a redraw function in the application program code. An application’s redraw function must be declared in the interface file, so that GIL knows to call it when a redraw is necessary. The syntax of a redraw function declaration is as follows:

    REDRAWFUNC (funcname )

The following parameter is mandatory:

    funcname : name - the name string of the function as declared in the FuncNames array.

4.2.11 Miscellaneous Parameter Declarations

The following are declarations which are not associated with a particular interface element. The HIGHLIGHT, SPEEDUP, and BUTTONSTUFF declarations are instead generally associated with multiple interface elements.

The HIGHLIGHT declaration determines the color used to highlight the current selection on a popup menu. Its syntax is:

    HIGHLIGHT (highlightcolor )

Its parameter is mandatory:

    highlightcolor : name - a color name or alias.

The highlight color in giltest is set to “aqua”:
HIGHLIGHT (aqua)

The SPEEDUP declaration is used for repeatable buttons (See Section 3.1). It specifies how quickly
the repetition rate should accelerate. The repetition rate is made deliberately slow to start off
with, to minimize the chance of an accidental repetition. As the button is held down, this rate
increases. This keeps the user from having to wait too long for repetitions to occur.

The syntax for a SPEEDUP declaration is as follows:

SPEEDUP ( speeduprate )

Its parameter is:

speeduprate : integer - how quickly the repetition rate for a repeatable button should increase.
    Usually, 100 is used for this value.

The SPEEDUP declaration for giltest is:

SPEEDUP (100)

The BUTTONSTUFF declaration sets certain global button characteristics. Its syntax is:

BUTTONSTUFF ( bordercolor diagonals? highlight_TL highlight_BR )

All of the parameters for BUTTONSTUFF declarations are mandatory. The parameters are:

bordercolor : name - the color of button borders.
diagonals? : integer - non-zero if and only if each disabled button should have a diagonal
    line drawn across it.
highlight_TL : name - the color of the highlight on the top and left edges of each button.
    See Section 3.1 regarding button depressing visuals.
highlight_BR : name - the color of the highlight on the bottom and right edges of each
    button.

The BUTTONSTUFF declaration for giltest is:

BUTTONSTUFF (black 0 white black)
5 GIL Image Format

GIL picture functions operate on runlength encoded image files. Runlength encoding is a form of image compression. Instead of representing an image as a two-dimensional array of pixel values, runlength encoding represents each row of the image as a series of runs. A run consists of several adjacent pixels with the same value. A run is represented in the image file as a record with two fields:

| run length | pixel value |

The run length field contains the number of pixels in the run. The pixel value field contains the common value of the pixels in the run.

All runlength encoded image files have the following format:

| tag | row count | column count | row vector | runs |

The 4 byte tag field identifies the image type. GIL supports four types of run length encoding, differentiated by the number of bytes per run, and the number of bits per pixel value field:

1. 1 byte per run. The high order 4 bits are the run length, and the low order 4 bits are the pixel value. Typically the pixel value is the color value for a 16-color palette. A zero (0) in the runlength field represents a runlength of 16.

2. 2 bytes per run. The first byte is the run length, and the second byte is an 8 bit pixel value. The pixel value can be an 8 bit palette index. Alternatively, it can be a data value which is converted to a color index by an application program function that is passed to GSdrawpict or GSdrawfilepict as a parameter. A zero (0) in the runlength field represents a runlength of 256.

3. 2 bytes per run. The high order 4 bits are the run length, and the low order 12 bits are the pixel value. The pixel value is always a data value, and must be converted to a color by the application program. A zero (0) in the runlength field represents a runlength of 16.

4. 3 bytes per run. The first byte is the run length, and the next two bytes are a 16 bit pixel value. The pixel value is always a data value, and must be converted to a color by the application program. A zero (0) in the runlength field represents a runlength of 256.

The row count and column count fields contain the number of pixel rows and pixel columns, respectively.

The row vector indicates the position in the file for the beginning of each row. This allows direct access to each row of the image (to get to a pixel within a row, the row must be processed in order). rowvector[0] stores the actual byte location within the file of the first run of row 0. Each item of
rowvector thereafter stores the offset from the previous row. This allows the row vector values to be only 2 bytes long, without limiting the entire image file size.

The runs section of the file contains the pixel runs for the image.

5.1 Runlength Encoder

GIL includes a utility program called reencode which converts a raster image file to GIL’s runlength encoded format. Currently, reencode can only convert (uncompressed) TIFF files and raw raster files, but we may add other image formats in the future. Image files created with reencode on any GIL-supported platform can be ported to any other GIL-supported platform without modification. Use reencode as follows:

reencode bits-per-pixel infile outfile

bits-per-pixel indicates which of the four runlength encodings will be used. The acceptable values for bits-per-pixel and the corresponding sizes for the run length and pixel value fields are as follows:

<table>
<thead>
<tr>
<th>bits-per-pixel</th>
<th>run length field</th>
<th>pixel value field</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>

infile is the input file in TIFF or raw raster format. It should contain a series of pixel values of the appropriate format. If bits-per-pixel is 4, then input pixels should be one byte per pixel with only the lower 4 bits significant. If bits-per-pixel is 8, then input pixels should be one byte per pixel, and all 8 bits will be treated as significant. If bits-per-pixel is 12, then input pixels should be two bytes per pixel, with only the lower 12 bits significant. If bits-per-pixel is 16, then input pixels should be two bytes per pixel, and all 16 bits will be treated as significant.

outfile is the name of the resulting runlength encoded image. The outfile argument can be omitted, in which case the name of the runlength encoded image file will be derived from the name of the raster image file by replacing the extension of the raster image file name with the extension “.rlX”, where X is the least significant decimal digit of bits-per-pixel. (“rlX” is simply appended to the end of the raster image file name if it has no extension.)

For certain pixel configurations, the application programmer may have a choice in which runlength encoding format to use. For example, raster images with 4 significant bits per pixel can be encoded with bits-per-pixel as either 4 or 8. If bits-per-pixel is set to 4, then the runlength codes will be shorter (only one byte), but the maximum runlength will be 16. Alternatively, if bits-per-pixel is set to 8, then the runlength codes will be longer (two bytes), but the maximum runlength can now be 256. Runlength encoded images that yield long runlengths may be smaller with bits-per-pixel set to 8; programmers may wish to experiment. Likewise, images with 12 significant bits per pixel can be encoded with 12 or 16 bits per pixel.
6 Portability Issues

The concern for portability has driven several design decisions. Many portability problems are solved by masking platform dependencies with GIL API functions. Other portability problems are not so easily solved.

Graphics Resolutions The GIL API uses up to a 640x480 pixel portion of the screen, which is the entire screen on MS-DOS (VGA) and some Macintosh machines. Whereas MS-DOS applications control the entire screen, on the Macintosh the top of the screen is normally taken by a standard menu bar. To obtain 640x480 pixels we eliminate this bar from the Macintosh screen on 640x480 displays. On larger monitors, the GIL application works properly within a Macintosh OS window, with the Macintosh menu in its usual location at the top of the screen. We make little pretense of following Macintosh (or any other) GUI standards.

Colors We only require 16 colors for existing GeoSim modules. However, we anticipate future need for more than 16 but no more than 256 colors. Although this is not a problem on the Macintosh or X-Windows platforms, there is no standard among the many manufacturers of MS-DOS video components that provides 640x480 pixels with 256 colors. Despite this, we are able to use most of the major video hardware that does support this resolution with public domain graphics drivers.

File and Path Names. File names follow the MS-DOS convention (eight characters plus a three character extension), since the other platforms allow this format. Full and partial pathnames can be used, but pathnames should always use the Unix path separator: ‘/’. GIL translates this path separator to the platform’s native format.

Memory Issues. MS-DOS uses a segmented memory scheme which requires that memory blocks be confined to 64 Kilobytes. Additionally, MS-DOS does not provide virtual memory. Programmers who expect their GIL-based applications to run under MS-DOS should be aware of these facts, and should also restrict their total run-time memory requirements to less than 640 Kilobytes.
<table>
<thead>
<tr>
<th>Constant</th>
<th>Semantics</th>
<th>Function</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAW</td>
<td>Draw button immediately after enabling it.</td>
<td>GEnablebutton</td>
<td>draw</td>
</tr>
<tr>
<td></td>
<td>Draw button immediately after disabling it.</td>
<td>GDisablebutton</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Immediately draw button on screen.</td>
<td>GVisiblebutton</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Immediately erase button from screen.</td>
<td>Ginvisiblebutton</td>
<td></td>
</tr>
<tr>
<td>NO_DRAW</td>
<td>Wait until window that contains button is (re)drawn to (re)draw button.</td>
<td>GEnablebutton</td>
<td>draw</td>
</tr>
<tr>
<td></td>
<td>Wait until window which contains button is (re)drawn to erase button.</td>
<td>GDisablebutton</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GVisiblebutton</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSTORE</td>
<td>Enable backing store for the application.</td>
<td>GSInterfaceinit</td>
<td>bstore</td>
</tr>
<tr>
<td></td>
<td>Save screen when named window is drawn.</td>
<td>GDrawnamedwindow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Save screen when Alert window is drawn.</td>
<td>GAlert</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Save screen when Help window is drawn.</td>
<td>GDohelpscreens</td>
<td></td>
</tr>
<tr>
<td>NO_BSTORE</td>
<td>Disable backing store for the application.</td>
<td>GSInterfaceinit</td>
<td>bstore</td>
</tr>
<tr>
<td></td>
<td>Don’t save screen when named window is drawn.</td>
<td>GDrawnamedwindow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Don’t save screen when Alert window is drawn.</td>
<td>GAlert</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Don’t save screen when Help window is drawn.</td>
<td>GDohelpscreens</td>
<td></td>
</tr>
<tr>
<td>POPUP</td>
<td>Deactivate all other active windows when named window is drawn.</td>
<td>GDrawnamedwindow</td>
<td>popup</td>
</tr>
<tr>
<td>NO_POPUP</td>
<td>Keep all other active windows active when named window is drawn.</td>
<td>GDrawnamedwindow</td>
<td>popup</td>
</tr>
<tr>
<td>SMALL</td>
<td>Use small font for text.</td>
<td>GSetfont</td>
<td>size</td>
</tr>
<tr>
<td>LARGE</td>
<td>Use large font for text.</td>
<td>GSetfont</td>
<td>size</td>
</tr>
<tr>
<td>THIN</td>
<td>Use thin line width.</td>
<td>GSetlinesize</td>
<td>size</td>
</tr>
<tr>
<td>THICK</td>
<td>Use thick line width.</td>
<td>GSetlinesize</td>
<td>size</td>
</tr>
<tr>
<td>CURR_MSG</td>
<td>Show current help message.</td>
<td>GDisplayhelpmessage</td>
<td>messnum</td>
</tr>
<tr>
<td>PREV_MSG</td>
<td>Show previous help message.</td>
<td>GDisplayhelpmessage</td>
<td>messnum</td>
</tr>
</tbody>
</table>

Table 1: Named Constants used as Arguments to GIL Functions.

7 GIL Constants and Data Types

7.1 Constants

Some GIL functions expect named constants (or variables which evaluate to these constants) as arguments. These constants are shown in Table 1. Each constant name is listed in the column labeled Constant. The semantic significance of using a particular constant as an argument to a particular function can be found in the column labeled Semantics by looking across the row which contains the desired constant and function. The functions which use each constant are listed in the column labeled Function. For each function, the parameter for which the constant is a legal value is listed in the column labeled Parameter.

Key Constants Some keys generate different codes on different platforms. GIL converts each of these codes to a universal value which is consistent across all platforms, and returns this value to the application’s keyboard handler function. Each such value is represented by a defined constant
<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyBout</td>
<td>Backspace Key</td>
</tr>
<tr>
<td>KeyTab</td>
<td>Tab Key</td>
</tr>
<tr>
<td>KeyEnter</td>
<td>Return Key</td>
</tr>
<tr>
<td>KeyEscape</td>
<td>Escape Key</td>
</tr>
<tr>
<td>KeyUp</td>
<td>Up Arrow Key</td>
</tr>
<tr>
<td>KeyLeft</td>
<td>Left Arrow Key</td>
</tr>
<tr>
<td>KeyDown</td>
<td>Down Arrow Key</td>
</tr>
<tr>
<td>KeyRight</td>
<td>Right Arrow Key</td>
</tr>
</tbody>
</table>

Table 2: GIL Key Constants.

name. Each of these constants is shown in Table 2 alongside the key which produces it.

7.2 Data Types

7.2.1 Enumerated Types

Some GIL functions expect enumerated type values (or variables which evaluate to these values) as arguments. In some cases, GIL passes enumerated type values to application-defined functions. The GIL enumerated types and their values are shown in Tables 3 and 4. Each enumerated type name is listed in the column labeled Type. The legal values for each enumerated type are list in the column labeled Values. The semantic significance of using a particular value as an argument to a particular function is can be found in the column labeled Semantics by looking across the row which contains the desired value and function. The functions which use each enumerate type are listed in the column labeled Function. For each function, the parameter for which the enumerated type is a legal value is listed in the column labeled Parameter.

7.2.2 Handles

A handle is a unique, system-assigned identifier for items which are managed jointly by the system and the application program. GIL uses two types of handles: GListHandle and GScListHandle. Handles of these types are returned to the application program by the function GScListSetup. Other scrolling list functions expect parameters of type GListHandle or GScListHandle.

7.2.3 Character Strings

GIL defines several character string types, each of a different size. The size of each string type is defined as a GIL constant. These types and their sizes are shown in Table 5. BigString is used for parameters in a number of GIL functions. The sizes shown represent the number of actual
<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
<th>Semantics</th>
<th>Function</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>HelpScreenPage</td>
<td>FirstPage</td>
<td>Show the first page of the help file.</td>
<td>GSdohelpscreens</td>
<td>pagecommand</td>
</tr>
<tr>
<td></td>
<td>NextPage</td>
<td>Go forward one page.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PrevPage</td>
<td>Go back one page.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CurrPage</td>
<td>Redisplay current page.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphicsModeType</td>
<td>G640x480x16</td>
<td>16 color graphics mode.</td>
<td>GSinterfaceinit</td>
<td>graphmode</td>
</tr>
<tr>
<td></td>
<td>G640x480x256</td>
<td>256 color graphics mode.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GraphicsFileType</td>
<td>NO_FILE</td>
<td>Don't dump screen.</td>
<td>GSdumpscreen</td>
<td>filetype</td>
</tr>
<tr>
<td></td>
<td>BMP_16</td>
<td>16 color Windows 3.1 bitmap format.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PS_GRAY</td>
<td>PostScript Level 2.0 greyscale format.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PS_COLOR</td>
<td>PostScript Level 2.0 color format.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>GIF_16</td>
<td>16 color GIF87a format.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DispType</td>
<td>Highlight</td>
<td>Highlight pick list item.</td>
<td>GSdrawlistitem</td>
<td>displaytype</td>
</tr>
<tr>
<td></td>
<td>Unhighlight</td>
<td>Remove highlight from pick list item.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ButtonStatusType</td>
<td>ENABLED</td>
<td>Button is enabled.</td>
<td>GSgetbuttonspecs</td>
<td>status</td>
</tr>
<tr>
<td></td>
<td>DISABLED</td>
<td>Button is disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INVISIBLE</td>
<td>Button is invisible.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TMP_DISABLED</td>
<td>Button is disabled for a popup window.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DragAreaStatusType</td>
<td>ENABLED</td>
<td>Drag area is enabled.</td>
<td>GSgetdragareaspecs</td>
<td>status</td>
</tr>
<tr>
<td></td>
<td>DISABLED</td>
<td>Drag area is disabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ControlGroupType</td>
<td>WINDOW</td>
<td>Last control element was button or drag area.</td>
<td>GSgetlastcontrol</td>
<td>control.type</td>
</tr>
<tr>
<td></td>
<td>MENU</td>
<td>Last control element was menu item.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NONE</td>
<td>Last mouse click did not access control element.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GSjustType</td>
<td>JUST_LEFT</td>
<td>Left-justify.</td>
<td>GSsclistsetup</td>
<td>just</td>
</tr>
<tr>
<td></td>
<td>JUST_CENTER</td>
<td>Center list items.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JUST_RIGHT</td>
<td>Right-justify.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Enumerated types used as arguments to **GIL** functions.
7.2 Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
<th>Semantics</th>
<th>Function</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReaderStatusType</td>
<td>READ_INIT</td>
<td>Signals reader_func to do initialization.</td>
<td>reader_func in GSgenreader</td>
<td>status</td>
</tr>
<tr>
<td></td>
<td>READ_PROCESS</td>
<td>Signals reader_func to process a record.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ_FINAL</td>
<td>Signals reader_func to do post-processing.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DragStatusType</td>
<td>DRAG_INIT</td>
<td>mouse drag started.</td>
<td>func associated with a dragarea</td>
<td>status</td>
</tr>
<tr>
<td></td>
<td>DRAG_PROCESS</td>
<td>mouse drag in progress.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DRAG_FINAL</td>
<td>mouse drag ended.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOUSE_INSIDE</td>
<td>mouse inside drag area.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOUSE_OUTSIDE</td>
<td>mouse out of drag area.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MOUSE_CLICK</td>
<td>mouse clicked in drag area.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Enumerated types used as arguments to application-defined functions.

<table>
<thead>
<tr>
<th>String Type</th>
<th>Size</th>
<th>Size Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileName</td>
<td>13</td>
<td>FILENAMELEN</td>
</tr>
<tr>
<td>GSString</td>
<td>40</td>
<td>GSSTRINGLEN</td>
</tr>
<tr>
<td>BigString</td>
<td>80</td>
<td>BIGSTRINGLEN</td>
</tr>
<tr>
<td>HugeString</td>
<td>255</td>
<td>HUGESTRINGLEN</td>
</tr>
</tbody>
</table>

Table 5: GIL String Types.

characters; an extra position is reserved for the NULL character.

7.2.4 Boolean Values

GIL defines a type named bool. The legal values for the bool data type are TRUE and FALSE. These values have the usual Boolean significance.

7.2.5 GColor

An application’s colors are set in the interface file. These colors can be referenced in the application code using variables of the GIL data type GColor. The GIL functions GSgetfieldrect, GSgetlistcolor, GSgetpoint, GSgetcolor, GSgetdragareaspect, and GSconvertcolor can be used to assign values to variables of type GColor; these variables can then be used as arguments to GSsetcolor, which sets the global draw color.
7.2.6 FormatList

The GIL type FormatList is an array of character strings. It is used by GSrdformat to return a list of format strings from a GIL ASCII tagged database. Figure 3 shows a database file which includes the following format strings:

```
format=
x y pop low hi name labx laby	tamrof
```

GSrdformat reads the format strings and inserts them into a variable of type FormatList declared by the application programmer. See the description of GSrdformat in Section 8.4.3 for more information.

7.2.7 GSPicture

GSPicture is a structure which represents a runlength encoded picture. The application code does not set any of the fields in a variable of type GSPicture. Instead, a call to GSinitpic is used to initialize a GSPicture variable from a file. A picture represented by a variable of type GSPicture can be drawn by passing the variable to GSdowndrawpict. The GSPicture data type is also used by GSgetpixel, which returns specific pixel values from a runlength encoded picture.

7.2.8 IntrFunction

The GIL type IntrFunction is used to link a function name declared in an interface file to a function pointer declared in the source code. (See Section 3.14 for a description of function name-pointer linkage.) It is declared as follows:

```
typedef struct {
    char  *name;
    bool (*funcptr)();
} IntrFunction;
```

Initialize the name field to the name declared in the interface file; set the funcptr field to the address of the function. Figure 2 shows an example of an IntrFunction initialization.

The library routines refer to a global array of type IntrFunction called FuncNames. The FuncNames array must be declared by the application programmer. Usually, the FuncNames array is the only data item of type IntrFunction.
7.2.9 ControlType

The **ControlType** structure is used to pass information regarding the last (i.e., most recent) control element accessed by the user. A control element is an interface element which can be used to control the behavior of the application at runtime. Examples of control elements are buttons, menu items, and drag areas. A label is an example of an interface element which is not a control element.

This information can be useful in altering the functionality of a program. A walk-through tutorial is an example of a situation where an application programmer might want to do this. For a walk-through tutorial, an independent function is written that controls the tutorial. This function is activated and called each time through the event loop (i.e., each time there is user input). The tutorial function might turn off buttons that are usually on, for example, to prevent a user from pressing it until the tutorial instructs the user to do so.

The **GIL** function `GStlastcontrol` returns the **ControlType** structure for the most recently accessed control element. The parameter `control` is a pointer to a variable of type `ControlType`; it must be passed the address of a `ControlType` variable that is declared in the application source code. The type declaration for `ControlType` is:

```c
typedef struct {
    ControlGroupType type;
    BigString gname;
    BigString time;
    union {
        BigString name;
        int number;
    } control;
} ControlType;
```

The **type** field describes the type of control last accessed: a control located in a window, a control located in a menu (a menu item), or none. The **gname** field contains the name of the particular window or menu where the most recently accessed control resides. The **time** field gives the time that the control was accessed. Finally, the **control union** describes the name of the control if it is part of a window, or the **number** if it is a menu item.
8 GIL Function Library

This section contains descriptions for every function in GIL. Note that GIL functions which draw interface elements (those in Section 8.2) and pictures (those in Section 8.3.3) may change the state of graphics attributes. That is, the current window, color, font size of line thickness may have changed. Therefore, the application program should explicitly set such attributes before it uses text and graphics drawing functions (those in Section 8.3).

8.1 Initialization, Start-Up and Shutdown

8.1.1 Interface Initialization

Any program which uses GIL will need to call GSInterfaceInit to initialize the user interface. GSInterfaceInit creates the interface elements and opens the error log file. Functions for interface handling, manipulating interface elements, and text and graphics output, should not be called before GSInterfaceInit is called. It is best to call GSInterfaceInit from main, since GSInterfaceInit requires the arguments passed into main.

```c
void GSInterfaceInit (BigString appl_name, BigString ifile, int bstore,
                      char *elog_file_mode, GraphicsModeType graphmode,
                      int argc, char **argv);
```

Initializes the user interface.

- **appl_name** - the name of the application. This name is used to label the application’s window and icon in GIL versions that use them.
- **ifile** - the name of the interface file.
- **bstore** - BSTORE if backing store is to be used; NO_BSTORE otherwise.
- **elog_file_mode** - “w” if error.log should be overwritten each time the program is started; “a” if error messages should be appended to the end of the file.
- **graphmode** - constant G640x480x16 for 16 colors or constant G640x480x256 for 256 colors.
- **argc** - the value passed into main giving the number of command-line arguments in the command to start the application.
- **argv** - the value passed into main giving the command-line arguments used in the command to start the application.

8.1.2 Event Processing

GIL is responsible for processing the events associated with its interface elements. For example, a button labeled “Help” might cause a pop-up help window to appear. GIL links the user’s action to the program’s response, which may be defined in GIL, or may be defined by the application programmer.
void GSinterface(void);

Starts the interface handler (i.e., the event loop) and initializes the mouse. The interface handler then runs until stopped by GSquit or the application program exits. GSinterface does not return to its caller.

void GSdoeventsandreturn(void);

Starts the interface handler (i.e., the event loop). The interface handler then runs until there are no events to process, at which point control is returned to the calling function. Mouse initialization is not done in GSdoeventsandreturn, since it is generally called multiple times. Use GSmouseinit to initialize the mouse prior to the first call to GSdoeventsandreturn.

void GSmouseinit(void);

Initializes the mouse.

8.1.3 Shutdown

The function GSquit is called to shutdown GIL. The GIL event loop runs until GSquit is called. GSquit performs platform-specific clean-up, such as closing windows, or exiting graphics mode in MS-DOS. GSquit does not return to its caller.

void GSquit (char *format, ...);

Quit the program and write a message to stdout and error.log. The arguments of this function are like those of printf.
8.2 Functions for Manipulating Interface Elements

8.2.1 Functions for Window Support

Windows are declared in the interface file, as described in Section 4.2.2. The following functions are for manipulating windows.

```c
void GSdrawnamedwindow (char *name, int bstore, int popup);

Draws the named window on the screen, and makes it the current window. Do not use GSdrawnamedwindow to redraw a displayed window (use GSredrawnamedwindow instead).

name - the name of the window, as specified in the interface file.

bstore - BSTORE if backing store is to be used to save the screen area overdrawn by the window; NO_BSTORE otherwise. An error will result — causing the program to exit — if this window’s backing store is currently being used for another window.

popup - POPUP if this is to be treated as a popup window; NO_POPUP otherwise. A popup window deactivates other active windows when it is drawn. The other windows remain disabled until either the popup window is removed, or they are explicitly enabled by a call to GSsetactivenamedwindow in the application.
```

```c
void GSremovenamewindow (char *name);

Removes the named window from the screen, and makes “root” the current window.

name - the name of the window, as specified in the interface file.
```

```c
void GSsetcurrentnamedwindow (char *name);

Makes the named window the current window. GIL drawing functions send output to coordinates which are relative to the coordinates of the current window.

name - the name of the window, as specified in the interface file.
```

```c
void GSsetactivenamedwindow (char *name);

Activates the named window (without drawing it). A window must be active in order for users to use the interface elements (buttons, drag areas, etc.) which it contains.

name - the name of the window, as specified in the interface file.
```
8.2 Functions for Manipulating Interface Elements

void GSsetdeactivenamedwindow (char *name);

Deactivates the named window (without removing it from the screen). Deactivating a window
makes its buttons and drag areas unavailable to the user.

name - the name of the window, as specified in the interface file.

void GSremovetopwindow (void);

Removes the top (i.e., the current) window from the screen, and makes “root” the current
window.

void GSredrawnamedwindow (char *name);

Redraws the named window, and makes it the current window.

name - the name of the window, as specified in the interface file.

void GSredrawactivewindows (void);

Redraws all active windows.

char *GSgetcurrentwindowname (char *name);

Returns in name the name of the current (top) window. The return value is also name.
Application programmers must make sure that name points to sufficient memory to
store the window’s name string.

name - the name of the window, as specified in the interface file.

void GSgetwindowspecs (char *windowname, int *x, int *y, int *wd, int *ht, int *active);

Returns the position and dimensions of a named window.

windowname - the name of the window.

x - returns the x-coordinate of the left edge of the window, relative to the root window. If
x-coordinate is not required, you may pass NULL.

y - returns the y-coordinate of the top of the button, relative to the root window. If y-
coordinate is not required, you may pass NULL.

wd - returns the width (in pixels) of the window. If width is not required, you may pass NULL.
ht - returns the height (in pixels) of the window. If height is not required, you may pass NULL.

active - returns TRUE if window is active, and FALSE if window is inactive. If active status is not required, you may pass NULL.

void GSsetwindowspecs (char *wname, int *x, int *y, int *wd, int *ht);

Sets the origin (absolute) and dimensions of a window. The origin of a window is its top left corner. To leave a window dimension unchanged, pass NULL as the dimension's parameter.

wname - the name of the window, as specified in the interface file.

x - the x-coordinate of the desired origin.

y - the y-coordinate of the desired origin.

wd - the new width of the window.

ht - the new height of the window.

### 8.2.2 Functions for Button Support

void GSendablebutton (char *windowname, char *buttonname, int draw);

Enables a named button. A button must be enabled for a mouse click to affect it. Makes windowname the current window. If draw == DRAW, then the button is (re)drawn (with the enabled button text color).

windowname - the name of the window that contains the button.

buttonname - the name of the button to enable.

draw - DRAW to (re)draw, NO_DRAW otherwise.

void GSdisablebutton (char *windowname, char *buttonname, int draw);

Disables a named button. Users can see, but cannot press disabled buttons. Makes windowname the current window. If draw == DRAW, then the button is (re)drawn (with the disabled button text color).

windowname - the name of the window that contains the button.

buttonname - the name of the button to disable.

draw - DRAW to (re)draw, NO_DRAW otherwise.
void GSinvisibilitybutton (char *windowname, char *buttonname, int draw);

Makes a named button invisible (removes it from the screen). Previous state (enabled vs. disabled) is not remembered. Makes windowname the current window. If draw == DRAW, then the button is erased from the screen.

windowname - the name of the window that contains the button.
buttonname - the name of the button to make invisible.
draw - DRAW to (re)draw, NO_DRAW otherwise.

void GSvisibilitybutton (char *windowname, char *buttonname, int draw);

Makes a named button visible and enabled. Makes windowname the current window. If draw == DRAW, then the button is (re)drawing.

windowname - the name of the window that contains the button.
buttonname - the name of the button to make invisible.
draw - DRAW to (re)draw, NO_DRAW otherwise.

void GSgetbuttonspects (char *windowname, char *buttonname, int *x, int *y, int *wd, int *ht, ButtonStatusType *status);

Returns the position, dimensions, and status of a named button.

windowname - the name of the window that contains the button.
buttonname - the name of the button.
x - returns the x-coordinate of the left edge of the button. If x-coordinate is not required, you may pass NULL.
y - returns the y-coordinate of the top of the button. If y-coordinate is not required, you may pass NULL.
wd - returns the width (in pixels) of the button. If width is not required, you may pass NULL.
ht - returns the height (in pixels) of the button. If height is not required, you may pass NULL.
status - returns the status (enabled, disabled, invisible) of the button. If status is not required, you may pass NULL.

void GSsetbuttonspects (char *windowname, char *buttonname, int *x, int *y, int *wd, int *ht, ButtonStatusType *status);

Sets the position, dimensions, and status of a named button. If a button characteristic should not change, pass NULL as the characteristic's parameter.

windowname - the name of the window that contains the button.
**buttonname** - the name of the button.

**x** - sets the x-coordinate of the left edge of the button. If x-coordinate should not change, pass **NULL**.

**y** - sets the y-coordinate of the top of the button. If y-coordinate pass **NULL**.

**wd** - sets the width (in pixels) of the button. If width should not change, pass **NULL**.

**ht** - sets the height (in pixels) of the button. If height should not change, pass **NULL**.

**status** - sets the status (**ENABLED, DISABLED, INVISIBLE**) of the button. If status should not change, pass **NULL**.

```c
void GSsetbuttontext (char *windowname, char *buttonname, char *text);
```

Sets the text displayed in a button. This function does not cause the button to be redrawn.

**windowname** - the name of the window that contains the button.

**buttonname** - the name of the button.

**text** - a pointer to the new button text. **GIL** makes its own copy of the text.

### 8.2.3 Functions for Menu Support

```c
void GScreatenamedpopup (char *name);
```

Draws a named popup menu, and temporarily makes “root” the current window. The former current window is restored as the current window when the menu is removed. Ordinarily, an application program does not call this function. Popup menus are associated with buttons in the interface file (see Section 4.2.3). When a button is pressed, **GIL** displays the appropriate popup menu.

**name** - the name of the popup menu, as specified in the interface file.

```c
void GSinvisiblemenuitem (char *menuname, char *itemname);
```

Flags a popup menu item to be hidden. The item will not appear the next time the menu is drawn (remaining items “move up” to fill the gap.) Note that although the position of remaining items will appear different on the screen, the item names do not change.

**menuname** - the name of the menu that contains the item.

**itemname** - the name of the item.
void GSvisiblemenuitem (char *menuname, char *itemname);

   Flags a hidden popup menu to be restored to visibility. The item will reappear the
   next time the menu is drawn. Note that although the position of remaining items
   will appear different on the screen, the item names do not change.
   
   menuname - the name of the menu that contains the item.
   itemname - the name of the item.

void GSendablemenuitem (char *menuname, char *itemname);

   Enables a popup menu item. The enabled menu text color declared in the interface
   file is used for the item when the menu is subsequently displayed. A menu item
   must be enabled for a user to select it.
   
   menuname - the name of the menu that contains the item.
   itemname - the name of the item.

void GSdisablemenuitem (char *menuname, char *itemname);

   Disables a popup menu item. The disabled menu text color declared in the interface
   file is used for the item when the menu is subsequently displayed. Users can see,
   but cannot select disabled menu items.
   
   menuname - the name of the menu that contains the item.
   itemname - the name of the item.

void GStogglecheck (char *menuname, char *itemname);

   Toggles a check mark on a checklist menu item. Check marks are bool indicators;
   toggling a visible check mark makes it invisible and vice versa. Note that the
   application programmer is responsible for keeping track of the current state of
   check marks and their semantics — this function only affects the appearance of a
   menu item on screen.
   
   menuname - the name of the menu that contains the item.
   itemname - the name of the item.

void GSetmenuorigin (char *menuname, int x, int y);

   Sets the origin (absolute) of a menu. The origin of a menu is its top left corner.
   
   itemname - the name of the menu, as specified in the interface file.
x - the x-coordinate of the desired origin.
y - the y-coordinate of the desired origin.

int GSgetmenuitemnum (char *menuname, char *itemname);

Returns the position of item itemname in menu menuname. The first item in a menu has position 0.
menuname - the name of the menu.
itemname - the name of the item.

char *GSgetmenuitemname (char *menuname, int itemnum);

Returns the name of the itemnum-th item in menu menuname. The first item in the menu is item number 0.
menuname - the name of the menu.
itemnum - the position of the item in the menu.

8.2.4 Functions for Drag Area Support

void GSenabledragarea (char *windowname, char *dragareaname);

Enables a named drag area. A drag area must be enabled for its function to be called.
Makes windowname the current window.
windowname - the name of the window that contains the drag area.
dragareaname - the name of the drag area to enable.

void GSdisabledragarea (char *windowname, char *dragareaname);

Disables a named drag area. The function associated with a disabled drag area is not called. Makes windowname the current window.
windowname - the name of the window that contains the drag area.
dragareaname - the name of the drag area to disable.
8.2 Functions for Manipulating Interface Elements

void GSgetdragareaspecs (char *windowname, char *dragareaname,
   int *x, int *y, int *wd, int *ht,
   DragAreaStatusType *status,
   GSColor *color);

Returns the position, dimensions, and status of a named drag area.
windowname - the name of the window that contains the drag area.
dragareaname - the name of the drag area.
x - returns the x-coordinate of the left edge of the drag area. If x-coordinate is not
   required, you may pass NULL.
y - returns the y-coordinate of the top of the drag area. If y-coordinate is not required,
   you may pass NULL.
wd - returns the width (in pixels) of the drag area. If width is not required, you may
   pass NULL.
ht - returns the height (in pixels) of the drag area. If height is not required, you may
   pass NULL.
status - returns the status (ENABLED, DISABLED) of the drag area. If status is not
   required, you may pass NULL.
color - returns the color of the dragarea. If color is not required, you may pass NULL.

void GSsetdragareaspecs (char *windowname, char *dragareaname,
   int *x, int *y, int *wd, int *ht,
   DragAreaStatusType *status,
   GSColor *color);

Sets the position, dimensions, and status of a named drag area. If a drag area character-
istic should remain unchanged, pass NULL as that characteristic’s parameter.
windowname - the name of the window that contains the drag area.
dragareaname - the name of the drag area.
x - sets the x-coordinate of the left edge of the drag area. If x-coordinate should not
   change, pass NULL.
y - sets the y-coordinate of the top of the drag area. If y-coordinate should not change,
   pass NULL.
wd - sets the width (in pixels) of the drag area. If width should not change, pass NULL.
ht - sets the height (in pixels) of the drag area. If height should not change, pass NULL.
status - sets the status (ENABLED, DISABLED) of the drag area. If status should not
   change, pass NULL.
color - sets the color of the dragarea. If color should not change, pass NULL.
8.2.5 Functions for Field Support

```c
void GSgetfieldirect (char *fieldname, int *x, int *y, int *wd, int *ht,
                      GSColor *color);
```

Returns the enclosing box and color for a named field. Except `fieldname`, all parameters may be `NULL`. A `NULL` parameter returns nothing.

- **fieldname** - the name of the field.
- **x**, **y** - returns the pixel coordinates of the upper left corner of the field.
- **wd** - returns the width of the field in pixels.
- **ht** - returns the height of the field in pixels.
- **color** - returns the color associated with the field.

```c
void GSsetfieldirect (char *fieldname, int *x, int *y, int *wd, int *ht,
                      GSColor *color);
```

Sets the enclosing box and color for a named field. If a characteristic should remain unchanged, pass `NULL` as the parameter for that characteristic.

- **fieldname** - the name of the field.
- **x**, **y** - sets the pixel coordinates of the upper left corner of the field.
- **wd** - sets the width of the field in pixels.
- **ht** - sets the height of the field in pixels.
- **color** - sets the color associated with the field.

```c
void GSwclearrect (int x, int y, int wd, int ht);
```

Clears a rectangle in the current window with window relative dimensions (`x`, `y`, `wd`, `ht`) (i.e., sets rectangle to background color). `x`, `y` are relative to the current window.

- **x**, **y** - the pixel coordinates of the upper left corner of the field.
- **wd** - the width of the field in pixels.
- **ht** - the height of the field in pixels.

```c
void GSframefield (char *windowname, char *fieldname,
                   GSColor tlcolor, GSColor brcolor);
```

Draws a "shadowed" box around the field `fieldname`. The frame is drawn in `windowname`. If `windowname` is `NULL`, the field is drawn in the current window. `tlcolor` specifies the color of the top and left sides of the frame; `brcolor` specifies the color of the bottom and right sides of the frame.
8.2 Functions for Manipulating Interface Elements

`windowname` - the name of the window in which to draw. If `NULL`, the current window is used.
`fieldname` - the name of the field to frame.
`tldcolor` - the color of the top and left sides of the frame.
`brcolor` - the color of the bottom and right sides of the frame.

```c
void GSframedragarea (char *windowname, char *dragareaname, 
                       GSColor tldcolor, GSColor brcolor);
```

Draws a “shadowed” box around the drag area `dragareaname` in window `windowname`.
`tldcolor` specifies the color of the top and left sides of the frame; `brcolor` specifies the color of the bottom and right sides of the frame.
`windowname` - the window containing the drag area.
`dragareaname` - the name of the drag area.
`tldcolor` - the color of the top and left sides of the frame.
`brcolor` - the color of the bottom and right sides of the frame.
8.2.6 Scrolling List Functions

**Slot Numbers** Items in scrolling lists are referenced by *slot numbers*. An item’s slot number is simply its position in the scrolling list. The first item in a scrolling list has slot number 0.

**Handles** Two types of handles are used to refer to scrolling lists: **GSsclistHandle** and **GSlistHandle**. Passing the wrong handle to a function is a common programming error. Be sure to pass the correct handle to **GIL** scrolling list functions.

```c
void GSsclistsetup (char *windowname, char *listaname,
                    int cols, char *fieldname,
                    int colgap, int rowgap,
                    GSjustType just,
                    char *(*label_func)(int slotnum),
                    void (*settext_func)(GSlistHandle list, int slotnum),
                    void (*in_func)(int slotnum),
                    void (*out_func)(void),
                    void (*click_func)(int slotnum),
                    char *thDaname,
                    GColor sbarcol,
                    GColor topleftcol, GColor botrightcol,
                    int numlistitems,
                    int curpos,
                    void (*dragged_func)(int newslot),
                    char *upbtn, char *downbtn,
                    GSlistHandle *newlist,
                    GSsclistHandle *newscrl);  
```

Registers the attributes and functions of a scrolling list.

- **windowname** - the name of the window that contains the drag area that contains the scrolling list.
- **dragareaname** - the name of drag area that contains the picklist.
- **cols** - the number of columns of items to display in the scrolling list.
- **fieldname** - the name of a field which describes the dimensions of the topmost item slot in the list drag area. The field must be declared by the application programmer in the interface file. The location of the field is relative to the location of the drag area.
- **colgap** - the space, in pixels, between columns.
- **rowgap** - the space, in pixels, between rows.
- **just** - the justification of list items. Use **JUST_LEFT** for left justification, **JUST_CENTER** for center justification, and **JUST_RIGHT** for right justification.
8.2 Functions for Manipulating Interface Elements

label_func - a pointer to a function — supplied by the application programmer —
that returns a pointer to the text of the item in slot slotnum of the current page.
settext_func - a pointer to a function — supplied by the application programmer —
that sets the text attributes, such as color and font size, for an item.
in_func - a pointer to a function — supplied by the application programmer — that
is called when the mouse cursor is inside the drag area. Pass NULL if no in_func
is supplied.
out_func - a pointer to a function — supplied by the application programmer — that
is called when the mouse cursor moves outside the drag area. Pass NULL if no
out_func is supplied.
click_func - a pointer to a function — supplied by the application programmer —
that is called when the user clicks the mouse within the drag area. Pass NULL if
no click_func is required.

thDAname - the name of the drag area that contains the scroll bar.
sbarcol - the color of the sliding “thumb” inside the scroll bar.
topleftcol - the shadow color for the top and left edges of the sliding “thumb.”
botrightcol - the shadow color for the bottom and right edges of the sliding “thumb.”
numlistitems - the total number of items in the scrolling list.
curpos - the initial topmost visible item in the scrolling list. To display the first item
in the list, pass 0 as this parameter.
draggedfunc - a pointer to a function — supplied by the application programmer —
that is called when the sliding “thumb” has been dragged to a new location. Pass
NULL if no dragged_func is supplied.
upbtn - the name of the button which causes the scrolling list to scroll up by one item.
downbtn - the name of the button which causes the scrolling list to scroll down by one
item.

newlist - the address of a G resultListHandle data item. The G resultListHandle referenced
by newlist will be assigned the handle of the new list.

newscrl - the address of a GS c scrollHandle data item. The GS c scrollHandle referenced
by newscrl will be assigned the handle of the new scrolling list.

void G Sdrawscclist ( GS c scrollHandle sc list);

Draws a scrolling list. This function should be called after the scrolling list has been
created, and anytime the application’s redraw function is called when a scrolling
list is visible.

c list - the handle of the scrolling list.
void GDrawlistitem (GListHandle list, Disptype displaytype, int itemnum);

   Displays the item itemnum in a picklist.
   list - the handle of the picklist.
   displaytype - Highlight to highlight the item; Unhighlight to remove highlighting
                 from the item; New to redraw the item.
   itemnum - the slot number of the item to draw.

void GHandlelist (GListHandle list, int x, int y,
    DragStatusType status);

   Processes mouse actions in a picklist. The function associated with the list drag area
   must call GHandlelist to process mouse actions.
   list - the handle of the picklist.
   x - the x-coordinate passed to the drag function.
   y - the y-coordinate passed to the drag function.
   status - the drag status passed to the drag function.

void GHandlesclist (GScclistHandle sclist, int x, int y,
    DragStatusType status);

   Processes mouse actions in a scrolling list’s scroll bar. The function associated with
   the scroll bar drag area must call GHandlesclist to process mouse actions.
   sclist - the handle of the scrolling list.
   x - the x-coordinate passed to the drag function.
   y - the y-coordinate passed to the drag function.
   status - the drag status passed to the drag function.

void GHandleupbutton (GScclistHandle sclist);

   Scrolls a list’s display up by one item. The function associated with the scrolling list’s
   “up” button should call GHandleupbutton.
   sclist - the handle of the scrolling list.

void GHandledownbutton (GScclistHandle sclist);

   Scrolls a list’s display down by one item. The function associated with the scrolling
   list’s “down” button should call GHandledownbutton.
sclist - the handle of the scrolling list.

void GSsetsclistnumitems (GSsclistHandle sclist, int numitems,  
   bool redraw);

Sets the number of items in a scrolling list.  
sclist - the handle of the scrolling list.  
numitems - the (new) number of items in the list.  
redraw - TRUE to redraw the scrolling list to reflect the new number of items; FALSE  
   to supress redrawing the scrolling list.

void GSsetsclist xpos (GSsclistHandle sclist, int newpos, bool  
   redraw);

Sets the first visible item in a scrolling list.  
sclist - the handle of the scrolling list.  
newpos - the slot number of the item which will be the first visible item.  
redraw - TRUE to redraw the scrolling list to reflect the new number of items; FALSE  
   to supress redrawing the scrolling list.

int GSgetsclistpos (GSsclistHandle sclist);

Returns the slot number of the first visible item in a scrolling list.  
sclist - the handle of the scrolling list.

void GSdismantlesclist(GSsclistHandle sclist);

Releases memory used by a scrolling list. After calling GSdismantlesclist, the han- 
   dles of the scrolling list are invalid.  
sclist - the handle of the scrolling list.

8.2.7 Color List Functions

GSColor GSgetlistcolor (int listnum, int colornum);

Returns the color at the requested index. The caller only needs to know the index for  
the color on the interface file’s color list.  
listnum - the number of the color list. The number of a colorlist is determined by  
   its position in relation to other color lists in the interface file; the first color list  
   declared in an interface file is number 0, the second is number 1, etc.  
colornum - the number of the color in the list. The number of a color is determined by  
   its relative position in the list; the first color listed is color 0, the second is color  
   1, etc.
8.2.8 Alert and Help Screen Display Utilities

void GSdisplayhelpmessage (int message);

Displays a message in the “Message” window. A window named “Message” must be
declared in the interface file.

message - the number of the message, as specified in the interface file. The constants
PREV_MSG and CURR_MSG can be used to redisplay the previous and current mes-
sages, respectively.

int GSgetcurrmgs (void);

Returns the number of the current message displayed in the “Message” window.

void GSalert (int helpmsg, int bstore, char *msgname, ...);

Draws window named “Alert” as a popup window and displays an urgent message in
it. The message is specified by an arbitrary number of string arguments. There
must be an empty string as the last argument. A window named “Alert” must be
declared in the interface file. The “Alert” window becomes the current window as
a result of this function.

helpmsg - the number of the message to be displayed in the “Message” window; “Mes-
sage” window must be declared in the interface file. The enumerated constants
PREV_MSG and CURR_MSG can be used to redisplay the previous and current mes-
sages, respectively.
bstore - BSTORE if backing store is to be used to save the screen area overdrawn by
the “Alert” window; NO_BSTORE otherwise.
msgname - the name of a message declared in the interface file. An arbitrary number
of messages may be displayed; the last string must be the empty string ("\").

bool GSdohelpscreens (BigString filename, HelpScreenPage pagecommand,
int bstore, int helpmsg);

Displays help text from a file in the popup “Help” window; the “Help” window must
be declared in the interface file. This function automatically splits the text into
pages of 25 lines each.
filename - the name of the file which holds the screen’s text.
pagecommand - FirstPage when “Help” window is first opened; NextPage to go for-
ward one page; PrevPage to go back one page; CurrPage to redraw current page.
bstore - BSTORE if backing store is to be used to save the screen area overdrawn by the “Help” window; NO_BSTORE otherwise.

helpmsg - the number of the message to be displayed in the “Message” window; “Message” window must be declared in the interface file. The enumerated constants PREV_MSG and CURR_MSG can be used to redisplay the previous and current messages, respectively.
8.3 Functions for Screen Text, Numbers, and Graphics

8.3.1 Text and Numeric Output Functions.

```c
void GSsetfont (FontSizeType size);

Sets the current font to either SMALL or LARGE.
size - the font size; either SMALL or LARGE.
```

```c
int GSsetwtextwidth (char *str);

Returns the width — in pixels — of str in the current font.
str - the string of text to measure.
```

```c
int GStextheight (char *str);

Returns the height — in pixels — of str in the current font.
str - the string of text to measure.
```

```c
void GSwwritemsg (int x, int y, char *msgtext);

Writes msgtext in the current window.
x - the x-coordinate of the screen pixel at which to write the message, relative to the
current window. The lower left corner of the message string will appear at this
pixel.
y - the y-coordinate of the screen pixel at which to write the message, relative to the
current window. The lower left corner of the message string will appear at this
pixel.
msgtext - the text of the message to be written.
```

```c
void GSwriteclippedmsg (int clipx, int clipy, int clipwd, int clipht, 
                        int msgx, int msgy, 
                        char *msgtext);

Writes text in the current window. The message is “clipped” to the box clipx, 
clipy, clipwd, clipht; that is, no part of the message will be drawn outside of 
the box.
clipx - the left side of the bounding box.
clipy - the top side of the bounding box.
clipwd - the width of the bounding box.
clipht - the height of the bounding box.
msgx - the x-coordinate of the screen pixel at which to write the message, relative to the current window. The lower left corner of the message string will appear at this pixel.
msgy - the y-coordinate of the screen pixel at which to write the message, relative to the current window. The lower left corner of the message string will appear at this pixel.
msgtext - the text of the message to be written.

void GSwritecommannum (int x, int y, double num);

Converts a double to a string with commas, and writes it to the current window.

x - the x-coordinate of the screen pixel at which to write the number, relative to the current window. The lower left corner of the number string will appear at this pixel.

y - the y-coordinate of the screen pixel at which to write the number, relative to the current window. The lower left corner of the number string will appear at this pixel.

num - the double to convert.

void GSwritereal (int wd, int precision, int x, int y, double num);

Writes num to the current window at position (x, y).

wd - minimum number of characters to print, left padded with blanks.

precision - number of digits to print after the decimal point.

x - the x-coordinate of the screen pixel at which to write the number, relative to the current window. The lower left corner of the number string will appear at this pixel.

y - the y-coordinate of the screen pixel at which to write the number, relative to the current window. The lower left corner of the number string will appear at this pixel.

num - the real number to be written.

void GSwriteint (int wd, int x, int y, long num);

Writes num to the current window at (x, y).
wd - minimum number of characters to print, padding with blanks.

x - the x-coordinate of the screen pixel at which to write the number, relative to the current window. The lower left corner of the number string will appear at this pixel.

y - the y-coordinate of the screen pixel at which to write the number, relative to the current window. The lower left corner of the number string will appear at this pixel.

num - the integer to be written.

char *GSgetmsg (char *messagename);

Returns the text of the named message. This text should not be modified by the application program.

messagename - the name of the message to retrieve, as declared in the interface file.

void GSwritenamedmsg (int x, int y, char *messagename);

Puts named text message in a window. Named messages are declared in the interface file.

x - the x-coordinate of the screen pixel at which to write the message, relative to the current window. The lower left corner of the message string will appear at this pixel.

y - the y-coordinate of the screen pixel at which to write the message, relative to the current window. The lower left corner of the message string will appear at this pixel.

messagename - the name of the message to display.

8.3.2 Functions for Graphics Support

void GSsetcolor (GSColor color);

Sets global draw color for lines, text, fills.

color - the color to draw in.

GSColor GSgetcolor (void);

Gets current global draw color for lines, text, fills.
void GSsetlinesize (int size);

Sets the thickness for all future lines to "size".

size - the thickness desired, in pixels. Two sizes are supported. Each size is represented by a GIL constant (shown in parentheses): 1-pixel (THIN) and 3-pixel (THICK).

void GSdrawline (int x0, int y0, int x1, int y1);

Draws a line of current color with current thickness in the current window from \((x_0, y_0)\) to \((x_1, y_1)\). Coordinates \(x_0, y_0, x_1, y_1\) are relative to the current window. \(x_0, y_0, x_1, y_1\) - the endpoints of the line.

void GSdrawrect (int x, int y, int wd, int ht);

Draws a frame rectangle of current color with dimensions \((wd, ht)\) at location \((x, y)\), relative to the current window.

\(x, y, wd, ht\) - the location and dimensions of the rectangle, relative to the current window.

void GSfillrect (int x, int y, int wd, int ht);

Draws a solid rectangle in the current window with dimensions \((wd, ht)\) at location \((x, y)\), relative to the current window.

\(x, y, wd, ht\) - the location and dimensions of the rectangle, relative to the current window.

void GSfillpoly (int numpoints, int *points);

Draws and fills a polygon using current line style and color.

numpoints - the number of vertices in the polygon

points - an array that contains the \(x\) and \(y\) coordinates of each vertex in the polygon.

The coordinates alternate \(x_0, y_0, x_1, y_1, \ldots\) in the array. The array contains exactly \(2 \times numpoints\) points.

void GSdrawpoint (int x, int y);

Sets a pixel in the current window to current color. The current color is a global attribute set by GSsetcolor.
x - the x-coordinate of the point, relative to the left edge of the current window.

y - the y-coordinate of the point, relative to the top of the current window.

GSColor GSwgetpoint (int x, int y);

Gets the color value of a pixel in a window.

x - the x-coordinate of the point, relative to the left edge of the current window.

y - the y-coordinate of the point, relative to the top of the current window.

void GSwdrawellipse (int x, int y, int wd, int ht);

Draws an ellipse of current color with dimensions (wd, ht) centered at (x, y). The
coordinates x and y are relative to the current window.

x, y - the center of the ellipse, relative to the current window.

wd, ht - the dimensions of the ellipse.

void GSwfillellipse (int x, int y, int wd, int ht);

Draws and fills an ellipse of current color with dimensions (wd, ht) centered at (x, y). The coordinates x and y are relative to the current window.

x, y - the center of the ellipse, relative to the current window.

wd, ht - the dimensions of the ellipse.

void GSwdrawxorbox (int x, int y, int wd, int ht);

Draws an XOR box. An XOR box is a frame rectangle which is one pixel in thickness.
The value of each pixel in the rectangle is obtained by taking the exclusive or of the
different pixel’s previous value with the value for white. This makes it very likely that
each pixel in the rectangle will stand out from those around it. To “erase” an
XOR box, simply call GSwdrawxorbox with the arguments used to draw the box.
XOR boxes are commonly used in GUI systems to show the outline of an object as
it is being dragged. This is accomplished in GIL by using drag functions: start by
drawing the XOR box at the location of the object on a DRAG_INIT, then erasing
it and redrawing it in the new location on a DRAG_PROCESS, and finally erasing
the XOR box and drawing the object at the new location on a DRAG_FINAL.

x, y, wd, ht - the location and dimensions of the XOR box, relative to the current
window.
void GSdrawinvertedbox (int x, int y, int wd, int ht);

Draws a box by inverting the pixels under the box.
x, y, wd, ht - the location and dimensions of the inverted box, relative to the current window.

void GSdrawxorline (int x0, int y0, int x1, int y1);

Draws a line in XOR mode (see above) from (x0, y0) to (x1, y1). Coordinates x0, y0, x1, y1 are relative to the current window.
x0, y0, x1, y1 - the endpoints of the line.

void GSdrawinvertedline (int x0, int y0, int x1, int y1);

Draws a line by inverting the pixels under the line from (x0, y0) to (x1, y1). Coordinates x0, y0, x1, y1 are relative to the current window.
x0, y0, x1, y1 - the endpoints of the line.

void GSfastrow (int x, int y, GSColor row[], int n);

Draws a row of n pixels beginning at (x, y). The array row must be initialized by the application to contain the desired color values of the pixels in the row. Color values can be obtained with the function GSconvertcolor (See page 68.)
x, y - the window-relative coordinates of the first pixel in the row.
row - an array of color index values for the pixels in the row.
n - the number of pixels in the row.

8.3.3 Functions for Picture Support

void GSdrawfilepict (FILE *fp, int x, int y, int wd, int ht,
int offx, int offy,
GSColor (*conversion_func)(unsigned short, int, int, int));

Reads a runlength encoded image from the file *fp, and displays the requested part on the screen.
fp - a pointer to the file that contains the image.
x, y, wd, ht - the location and dimensions of the image, relative to the current window.
offx, offy - the row and column in the image at which drawing should start. If offx is greater than 0, the left edge of the image will be cropped. If offy is greater than 0, the top edge of the image will be cropped.

conversion_func - a pointer to an application program function that converts a pixel's value in the image file into a color value.

bool GSinitpic (FILE *fp, GSPicture *pict, void *mem);

Reads a runlength encoded image into memory for future processing. Returns TRUE iff processing the image was successful.

fp - a pointer to the file from which to read the image
pict - the structure describing the picture; this is filled in by GSinitpic. Space must be provided by the application programmer.
mem - the memory area provided in which to store the image. The size of this area should at least (size of the file - 8) bytes.

void GSgetpixel (GSPicture *pict, int x, int y, unsigned short *buf, int n);

Returns in buf the n pixel values from the row beginning at (x, y) in picture pict.
pict - the descriptor for the image as created by GSinitpic.
x, y - the location of the beginning of the row from which to get pixel values.
buf - buffer in which pixel values are returned.
n - number of pixels' values to return.

void GSgetfilepixel (FILE *pictfile, int x, int y, unsigned short *buf, int n);

Returns in buf the n pixel values from the row beginning at (x, y) in the run length encoded image stored in file pictfile.
pictfile - an open file containing a run length encoded image (see Section 5).
x, y - the location of the beginning of the row from which to get pixel values.
buf - buffer in which pixel values are returned.
n - number of pixels' values to return.
void GSdrawpict (GSPicture *pict, int x, int y, int wd, int ht,
    int offx, int offy, int scale,
    GColor (*conversion_func)(unsigned short, int, int, int));

Draws the requested part of an in-memory runlength encoded image onto the screen.
pict - the descriptor for the image, initialized by GSinitpic.
x, y, wd, ht - the location and dimensions of the image, relative to the current
    window.
offx, offy - the row and column in the image at which drawing should start. If offx
    is greater than 0, the left edge of the image will be cropped. If offy is greater
    than 0, the top edge of the image will be cropped.
scale - determines the size of an input pixel in terms of screen pixels.
conversion_func - a pointer to an application program function that converts an
    image pixel value into a color value.

void GSgetfilepictspecs (FILE *pictfile,
    unsigned short *nrows, unsigned short *ncols);

Returns the number of rows and number of columns in the run length encoded image
    stored in file pictfile.
pictfile - an open file containing a run length encoded image (see Section 5).
nrows - a pointer to an unsigned short, where the number of rows in the picture will
    be stored.
ncols - a pointer to an unsigned short, where the number of columns in the picture
    will be stored.

8.3.4 Keyboard Support Utilities

void GScharfunc (char *name);

Activates a registered independent function to handle character events. Only one
    keyboard handler may be active at a time.
name - the name string for the function as declared in the FuncNames table. (The name
    string is the first field in the FuncNames declaration.) The function must also be
    registered via a FUNCTION declaration in the interface file.

void GSclearcharfunc (void);

Deactivates the active keyboard handler function (leaving no active keyboard handler).
8.3.5 Mouse Support Utilities

void GSsetmouseposition (int x, int y);

Moves the mouse cursor to \((x, y)\) on the screen. Has no effect in Macintosh version.

void GSmouseoff (void);

Hides the mouse cursor in MS-DOS. Has no effect on other platforms.

void GSmouseon (void);

Restores the mouse cursor in MS-DOS. Has no effect on other platforms.

8.3.6 Miscellaneous Support

void GSdologinscreen (char *prompt, int bstore,
            bool (*return_func)(void),
            GSColor login_text_color, BigString user_name);

Draws the login window and activates a built-in keyboard handler for a user login. This is a convenient way of getting the user’s name for program output, security checks, etc. The application programmer must declare a window named login and a field named login in the interface file. The login window is treated as a popup window. The login field is for user text entry; it is positioned relative to the login window. Only keyboard input received while login is the current window is processed.

prompt - a message displayed above the login field which prompts the user to type in his or her name. The message is drawn in the login window foreground color.

bstore - BSTORE if backing store is to be used to save the screen area overdrawn by the login window; NO_BSTORE otherwise.

return_func - when the user presses “Return”, the login window is removed from the screen, the keyboard handler is deactivated, and then return_func is executed. The application programmer supplies return_func to tell GIL what to do (e.g., draw another window) after removing the login window. This is mainly useful when the login window is the first one drawn; in this case, there will be nothing on screen that the user can access when the login window is removed. If other windows are drawn before GSdologinscreen is called, then the application programmer may want to make this argument a pointer to a null function — i.e., one with no executable statements — or pass a NULL pointer as the argument.

login_text_color - the color to use when echoing the user’s input.
user_name - a character buffer to store the user's login input. This must be a static variable; an automatic variable would be destroyed as soon as the function which called GSdologinscreen returned (GSdologinscreen returns before the actual login starts.) The application programmer is responsible for using the login input; GIL does not write it to any of the log files.

void GSdisplayversion (BigString versionfile);

Extracts the current version number of the program from a text file and displays it in a field named Version. A field named Version must be declared in the interface file.

versionfile - the file containing the current version number. The version number must appear in a line in the file which begins with the word Version.

long GSMemavail (void)

Returns the number of bytes of available memory.

void GSActivatefunction (char *name);

Activates a registered independent function to be called repeatedly from inside the event loop until GSdeactivatefunction is called.

funcname - the name string of the function to activate, as declared in the FuncNames table. (The name string is the first field in the declaration.) The function must also be registered via a FUNCTION declaration in the interface file.

void GSdeactivatefunction (char *name);

Deactivates a registered independent function that was previously activated.

funcname - the name of the function to deactivate (the same name that was used to activate the function).

void GSgetcursorxy (int *x, int *y);

Returns in (x, y) the current position of the cursor relative to the current window.

x, y - pointers to integers where mouse coordinates will be stored.
GColor GSconvertcolor (char *colormame);

Converts a color name to a color palette index.
   colormame - color name to be converted.

void GSsavewindows (FILE *fp);

Saves the status of the interface elements to a file. This includes the status (active or
inactive) of all windows, the status (enabled, disabled, or invisible) of all buttons,
the status (visible or invisible) of all popup menus, the status (enabled, disabled,
or invisible) of all popup menu items, and the status (checked or unchecked) of all
checklist menu check marks
   fp - file pointer for file in which to save status.

void GSloadwindows (FILE *fp);

Retrieves the status of the interface elements from a file. This includes the status
(active or inactive) of all windows, the status (enabled, disabled, or invisible)
of all buttons, the status (visible or invisible) of all popup menus, the status
(enabled, disabled, or invisible) of all popup menu items, and the status (checked
or unchecked) of all checklist menu check marks
   fp - file pointer for the file from which to load status.

bool GSdumpscreen (char *fname, GraphicsFileType filetype);

Dumps the current screen image to a file. An application calls GSdumpscreen with the
prefix of the filename that it wants the screen dumped to (e.g., "giltest") in fname.
The function attempts to open fname00.EXT (where EXT depends on filetype,
such as .bmp or .ps). If fname00.EXT already exists, it tries fname01.EXT, etc. up
to fname99.EXT.

Returns TRUE and the filename saved (in fname) on success. On failure, returns
FALSE.
   fname - filename prefix to which to save screen. fname may be a path or just a filename;
in either case, the actual file name should not exceed 6 characters to allow numeric
suffix. The name of the saved file is returned in fname, so the application needs
to allocate enough memory for the returned string (i.e., original length + 6).
   filetype - the graphics format in which the screen is saved. The following are valid
values:
      BMP_16 - 16 color Windows 3.1 bitmap format.
PS_GRAY - PostScript Level 2.0 greyscale format.
PS_COLOR - PostScript Level 2.0 color format.
GIF_16 - 16 color GIF87a format.

long GSfilesize (FILE *fp);
   Returns size in bytes of file fp.

void GSgetlastcontrol (ControlType *control);
   Returns the most recently accessed control element and it's attributes in control.
control - address of variable of type ControlType declared in application program.

void GSclearlastcontrol (void);
   Tells the event handler to forget the last control element accessed.

bool GShasFPU(void);
   Returns TRUE if machine has a floating point unit, otherwise returns FALSE.

void GSbell(void);
   Makes a short beep sound.

void GSclick(void);
   Makes a short click sound. This function has no effect in X Window versions.

void GSdelay(unsigned int delaytime);
   Causes the application to pause for delaytime milliseconds.
delaytime - number of milliseconds to pause.

bool GSbigendian (void);
   Returns non-zero if this machine is big-endian; returns 0 otherwise.

int GSnnumifuncs (IntrFunction func_array[]);
   Returns the number of functions in func_array. Implemented as a macro.
func_array - an array of function name to function pointer mappings (e.g., the Func-Names array).
8.4 Functions for File Access

8.4.1 Opening Other Files

FILE *GSopen(char *filename, char *mode);

Like fopen, but portable among all GIL environments.

filename - The name of the file to open.

mode_string - "r" to open for reading; "w" to truncate or create for writing; "a" to
open or create for writing at end of file; "r+" to open for reading and writing;
"w+" to truncate or create for reading and writing; "a+" to open or create for
reading and writing at end of file.

8.4.2 Log Files

The error log file (error.log) is initialized in GSInterfaceInit. GSinitlog must be called
to initialize the file output.log. This file can be used to record user input as well as
program output. GSinitlog returns FALSE if an error is encountered initializing output.log;
otherwise, it returns TRUE.

bool GSinitlog (char *mode_string);

Initializes the error log file.

mode_string - "w" if the output.log should be over written each time the program
is started; "a" if each user's output should be appended to the end of the file.

Sending Output to Log Files  The functions used to send output to the log files are mod-
eled closely after printf. They are wired directly to the files output.log and error.log,
respectively, so they do not require a file pointer argument, as does fprintf. GSelog re-
quires an integer code as an argument. If the code argument is 0, the program continues;
otherwise, the program exits with the argument as its error code.

void GSelog (char *format, ...);

Like printf. Writes to output.log.

void GSelog (int code, char *format, ...);

Like printf. Writes to error.log. If code is nonzero, the message is also written to
stdout, and the program exits with error code == code.
8.4 Functions for File Access

8.4.3 Database Reader Routines

```c
bool GSgenreader (FILE *fp,
    bool (*reader_func)(BigString keyword, int status,
    void *data),
    char *dbident, void *data)
```

This function is the driver for reading all GIL ASCII tagged databases. It is given a file pointer to the database file, an application-specific function (called the “local reader”) that is called when a keyword is read, a database identifier string, and a pointer to (optional) data space used by the local reader. GSgenreader then scans through the database looking for keywords. A keyword is any string of non-white space characters which ends in ‘=’. When a keyword is found, the local reader is called to process it. The local reader uses the library of generic reader I/O routines appearing below to access data values from the database. When the local reader returns, GSgenreader continues with the file pointer where the local reader left it. GSgenreader returns TRUE if successful, FALSE if there is an error.

- **fp** - file pointer to database file.
- **reader_func** - local reader function. The parameters of a local reader function are:
  - **keyword** - a keyword read from the database file. All keywords must end with ‘=’ (e.g., mykeyword=).
  - **status** - one of READ_INIT, READ_PROCESS, or READ_FINAL. GSgenreader always calls the local reader the first time with READ_INIT to allow it to do initialization, and the last time with READ_FINAL to allow it to do post-processing. When GSgenreader reads a keyword from the database file, it calls the local reader with READ_PROCESS, so the local reader can process the record which follows. Any READ_PROCESS calls come in between the READ_INIT call and the READ_FINAL call.
  - **data** - the same pointer (if any) supplied to the data parameter in GSgenreader. This parameter should be declared, even if it is not used.
- **dbident** - expected identifier tag for database. GSgenreader will return FALSE if the first keyword is not database=, or if the word immediately following it is not the string passed to dbident.
- **data** - pointer to a optional block of data used to communicate information from the local reader to the routine that calls GSgenreader. Use NULL if you don’t want to use this parameter.

Data Item Reader Functions The following functions are for reading a GIL ASCII tagged database. They should be called only from within a local reader.
bool GSrdstringeol (char *string, int size);

Reads and returns the rest of the current line (ignoring comments), compressing multiple spaces to one space. Return value is TRUE iff size characters or less are read; FALSE if EOF is encountered before size characters are read, or if more than size characters are read. (No more than size characters will be returned).

string - string read and returned to the calling procedure.
size - maximum length for string.

bool GSrdphrase (char *string, int size);

Reads and returns a string terminated by \n', \t', EOF, or '#', or multiple spaces (single spaces are returned in string.) Maximum string length is size. Return value is TRUE iff size characters are read; FALSE if EOF is encountered before size characters are read, or if more than size characters are read. (No more than size characters will be returned).

string - string read and returned to the calling procedure.
size - maximum length for string.

bool GSrdword (char *string, int size);

Reads and returns a string delimited by ' ', (space), \n', \t', EOF, or '#', with maximum length size. Equivalent to fscanf(GSDbfp, "%s", string), except that GSrdword won't read more characters than the size limit. Return value is TRUE iff size characters are read; FALSE if EOF is encountered before size characters are read, or if more than size characters are read. (No more than size characters will be returned).

string - string read and returned to the calling procedure.
size - maximum length for string.

long GSrdinteger (void);

Interprets the next string in the database file as a long integer, and returns this value.

double GSrdreal (void);

Interprets the next string in the database file as a double, and returns this value.
8.4 Functions for File Access

bool GSrdboolean (void);

Returns a bool value — TRUE if the word read is “TRUE” (ignoring case), else FALSE.

bool GSrdformat (int *numwords, FormatList keywords)

Reads the keywords following the “format” keyword and return these in an array. Keywords are scanned until the “tamrof” keyword is encountered. The rest of the line after “tamrof” is then skipped. Returns FALSE if there is a premature EOF. The list of keywords can be used by the local reader to process multiple records which have the same format before returning to GSgenreader. For example, the India city database shown in figure 3 contains records for 24 different cities. All of these records can be processed at one time by the local reader function based on the list of format strings that appears above them.

numwords - the number of format keywords read.
keywords - format keywords returned.

8.4.4 Opening Other Files

FILE *GSopen(char *filename, char *mode);

Like fopen, but portable among all GIL environments.
filename - The name of the file to open.
mode_string - “r” to open for reading; “w” to truncate or create for writing; “a” to open or create for writing at end of file; “r+” to open for reading and writing; “w+” to truncate or create for reading and writing; “a+” to open or create for reading and writing at end of file.
9  \textit{giltest}

\textit{giltest} is a program designed to demonstrate the functionality of many of the \textbf{GIL} routines. Users of \textit{giltest} can type input into a text field, or draw rectangles in a draw area. The message window at the bottom of the screen explains to users how to navigate through the program. \textit{giltest} is included with the \textbf{GIL} distribution. We recommend that you try running it, and follow its execution in the source code below.

9.1 Interface File for \textit{giltest}

\begin{verbatim}
PALETTE
# R G B
#
# ( black  0  0  0 )
# ( dgray  50 45 45 )
# ( mgray  30 30 55 )
# ( lgray  53 53 53 )
# ( aqua   0 50 50 )
# ( lblue  10 10 55 )
# ( peach  63 47 50 )
# ( blue   0  0 36 )
# ( green  0 36  0 )
# ( lpurple 63 50 63 )
# ( red    55  0  0 )
# ( yellow 63 63 21 )
# ( lcyan  50 59 63 )
# ( magenta 44  6 44 )
# ( sandy  63 51 51 )
# ( white  63 63 63 )
END #PALETTE

# Aliases for colors for ease of modification
# alias     color
#
# Window colors.
ALIAS ( rootbg  aqua )   # background color for root window
ALIAS ( rootfg  black )  # foreground color for root window
ALIAS ( wbdr    black )  # border color for windows
# Button colors.
ALIAS ( bbg     lblue )  # background color for buttons
ALIAS ( bef     white )  # text color for enabled buttons
ALIAS ( bdf     mgray )  # text color for disabled buttons
# Menu colors.
ALIAS ( popefg   black )  # text color for enabled popup menu items
ALIAS ( popdfrd  dgray )  # text color for disabled popup menu items
ALIAS ( popbg    peach )  # background color for popup menus
ALIAS ( popbdrd  black )  # border color for popup menus
# Color List colors.
ALIAS ( fillcolor0 black )  # first color in menu
ALIAS ( fillcolor1 yellow)  # second color in menu
ALIAS ( fillcolor2 white )   # third color in menu
ALIAS ( fillcolor3 lblue )   # fourth color in menu
\end{verbatim}
9.1 Interface File for giltest

ALIAS ( fillcolor4 red ) # fifth color in menu
     # Other colors.
ALIAS ( text   black ) # color of text in text entry field
ALIAS ( dragbg white ) # background color of draw dragarea

COLORLIST ( fillcolor0 fillcolor1 fillcolor2 fillcolor3 fillcolor4 )

HIGHLIGHT ( aqua )
SPEEDUP ( 100 )
BUTTONSTUFF ( black 0 white black )
FUNCTION ( intro 0 1 )
FUNCTION ( kbd_hndl 0 0 )
REDRAWFUNC ( redraw_func )

MSG ( hm10 |Look here for help messages as you go through the program.| )
MSG ( hm11 |Press "Again", or type a message, or press "Box Style" to| )
MSG ( hm12 |set parameters for drawing boxes in the Drawing Area.| )

MSG ( hm20 |Notice how the "Again" button has disappeared.| )
MSG ( hm21 |This was done with GS_invisiblebutton.| )
MSG ( hm22 |You can type a message if you like.| )

MSG ( hm30 |This is a regular menu.| )
MSG ( hm31 |It allows you to select a function.| )
MSG ( hm32 |Select a function, or click outside the menu to close it.| )

MSG ( hm40 |This is a checklist menu.| )
MSG ( hm41 |Each item on this menu represents a parameter.| )
MSG ( hm42 |Select one of the items and watch what happens to the menu.| )

MSG ( hm50 |A checkmark has appeared next to the selected item.| )
MSG ( hm51 |This program uses this to show that the corresponding parameter| )
MSG ( hm52 |has been turned on. Try drawing a box in the Draw Area.| )

MSG ( hm60 |The checkmark next to the selected item has disappeared.| )
MSG ( hm61 |This program uses this to show that the corresponding parameter| )
MSG ( hm62 |has been turned off. Try drawing a box in the Draw Area.| )

MSG ( hm70 |This is a checklist menu. Each item on this| )
MSG ( hm71 |menu represents a legal value for the FillColor parameter.| )
MSG ( hm72 |Select an unchecked item and watch what happens to the menu.| )

MSG ( hm80 |The checkmark has moved to the selected value.| )
MSG ( hm81 |Note that the code to remove the checkmark from the old value| )
MSG ( hm82 |is in the application program.| )

MSG ( hm90 |You selected the current value.| )
MSG ( hm91 |The checkmark appears to be unchanged, but it was really| )
MSG ( hm92 |removed by GUI, and then redrawn by the application.| )

MSG ( typedtoofar |You have reached the end of the text box.| )

POPUP ( funcs_menu 9 48 popefg popdfg popbg popbdr 0 )
     PLABEL ( ClearText 0 clear_text_func |Clear Text Field| )
     PLABEL ( ClearDraw 0 clear_draw_func |Clear Draw Area| )
     PLABEL ( QuitProg 1 quit_prog_func |Quit| )
POPPUP ( col_menu 9 198 popefg popdfg popbg popbdr 1 )
LABEL ( Black 1 set_color_func [Black] )
LABEL ( Yellow 1 set_color_func [Yellow] )
LABEL ( White 1 set_color_func [White] )
LABEL ( Blue 1 set_color_func [Blue] )
LABEL ( Red 1 set_color_func [Red] )

POPPUP ( box_menu 9 123 popefg popdfg popbg popbdr 1 )
LABEL ( DrawBorder 1 toggle_border_func [Draw Black Border] )
LABEL ( Fill 1 toggle_fill_func [Fill With Color] )

WINDOW ( root 0 0 640 480 white rootbg wbdr 2 1 0 )
BUTTON ( funcs |Functions| 9 23 105 25 befg bdfg bbg @funcs_menu 1 1 0 C )
BUTTON ( box |Box Style| 9 98 105 25 befg bdfg bbg @box_menu 1 1 0 C )
BUTTON ( color |Fill Color:| 9 173 105 25 befg bdfg bbg @col_menu 1 1 0 L )
LABEL ( 235 15 white |GeoSim Interface Library (GIL) Test| black )
LABEL ( 550 15 rootfg |GeoSim| )
FIELD ( version 450 5 60 19 rootfg )
FIELD ( fillcolor 69 178 10 10 rootfg )

WINDOW ( io 130 23 380 384 black mgray wbdr 2 1 0 )
BUTTON ( again |Hello Again| 220 75 105 25 befg bdfg bbg agn_func 1 1 0 C )
DRAGAREA ( dravarea 2 191 376 177 dragbg draw_func 1 )
FIELD ( hello1 220 70 121 16 yellow )
FIELD ( hello2 220 110 121 16 white )
FIELD ( dragfg 2 191 376 191 text )
FIELD ( dragbg 2 191 376 177 dragbg )
FIELD ( coordbg 2 370 120 12 white )
FIELD ( coordfg 2 380 378 12 black )
FIELD ( textbg 220 18 126 16 dgray )
FIELD ( textfg 222 30 120 370 text )
FIELD ( logo 20 20 155 155 white )
LABEL ( 135 380 rootfg |Drawing Area| )
LINE ( 0 368 380 368 2 black )
LINE ( 0 189 380 189 2 black )
RECT ( 2 370 376 12 white 2 fill )
RECT ( 218 16 130 20 dgray 2 fill )
RECT ( 218 16 130 20 wbdr 2 hollow )
LINE ( 218 35 347 35 1 white )
LINE ( 219 34 346 34 1 white )
LINE ( 347 35 347 16 1 white )
LINE ( 346 34 346 17 1 white )

WINDOW ( alert 145 150 350 100 popefg popdfg popbg popbdr 2 0 1 )
BUTTON ( OK |Acknowledged| 239 70 105 25 befg bdfg bbg done_func 1 1 0 C )

WINDOW ( message 5 428 630 39 black popbg wbdr 2 1 0 )
9.2 Source Code for giltest

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "geosim.h"

>Description:

```
#define CLEAR_DRAW_MENU_ITEM   "ClearDraw"
#define CLEAR_TEXT_MENU_ITEM   "ClearText"
#define FILL_COLOR_LIST        0
#define INITIAL_ITEM           0
#define TIME_TO_READ_MSG       2
#define FILL_COLOR_BUTTON_LABEL "Fill Color:"

//--- Help message numbers ---*
#define AGAIN_MSG        2
#define BOX_STYLE_CHECK_OFF_MSG 6
#define BOX_STYLE_CHECK_ON_MSG 5
#define BOX_STYLE_MENU_MSG  4
#define FILL_COLOR_MENU_MSG  7
#define INTRO_MSG        1
#define MENU_MENU_MSG     3
#define NEW_FILL_COLOR_MSG  8
#define SAME_FILL_COLOR_MSG 9

```

.Drawing Object declarations

```
typedef enum {
  rectObj, textObj
} ObjectType;

typedef struct {
  int x, y;
  char *str;
  GSCColor t_col;
} TextObj;

typedef struct {
  int x, y, wd, ht;
  bool fill, border;
  GSColor f_col, b_col;
} RectObj;

typedef struct DrawObj {
  ObjectType type;
  union {
    TextObj text;
    RectObj rect;
  } object;
  struct DrawObj *next_object;
```
} DrawObj;

/**-------------------------------------------------------------*/

void add_cursor_if_space (char *out_string, int field_wd);
DrawObj *add_rect (int x, int y, int wd, int ht,
    bool fill, bool border,
    GColor f_col, GColor b_col);
DrawObj *add_text (int x, int y, char *str, GColor t_col);
bool alert_done (void);
bool char_is_legal (char ch);
bool clear_draw (void);
bool clear_text (void);
bool do_again (void);
bool do_draw (int mouse_x, int mouse_y, DragStatusType status);
bool do_intro (void);
bool do_quit (void);
bool do_text (int mouse_x, int mouse_y, char ch);
void draw_all_objects (void);
void draw_object (DrawObj *object);
void init_pict (GSPicture *pict_ptr, FileName pict_file);
void init_colors (void);
GColor logo_converter (unsigned short code, int x, int y, int z);
bool set_color (char *item_name);
bool showfuncs_menu (void);
bool show_box_styles (void);
bool show_colors (void);
void show_fill_color (GColor);
bool toggle_border (char *item_name);
bool toggle_fill (char *item_name);
void write_hello (void);
void write_hello_again (void);
bool redraw_all (void);
void remove_all_objects (void);

/**-------------------------------------------------------------*/

bool AlertShowing = FALSE; /*--- Blocks typing when TRUE ---*/
bool DrawBorder = FALSE; /*--- Boxes get borders iff TRUE ---*/
bool FillBorder = FALSE; /*--- Boxes get filled iff TRUE ---*/
GColor FillColor; /*--- Color to fill boxes with ---*/
FileName GlobeFile = "logo.r14"; /*--- File that contains globe logo ---*/
/*FileName GlobeFile = "usmap.r14"; /*--- File that contains globe logo ---*/
GSPicture LOGoPicT; /*--- Structure for globe logo ---*/
BigString TextString; /*--- User’s input string ---*/
DrawObj *first_object, *last_object; /*--- Object list start and end ---*/
GColor aqua_color, mgray_color, white_color; /*--- Logo colors ---*/

/**-------------------------------------------------------------*/

Function Definitions

}
### 9.2 Source Code for giltest

```c
/*---------------------------------------------
add_cursor_if_space
    Add a text cursor ('_') to end of output string if there is enough
    space (in memory and on screen) for it.
---------------------------------------------*/
void add_cursor_if_space(
    char *out_string, /**< output string */
    int field_wd    /**< width of output field */
)
{
    /**< if space in memory for text cursor, add it to temp buffer */
    if (strlen(out_string) < BIGSTRINGLEN)
    {
        strcat(out_string, "_.");
        /**< if not enough space on screen for text cursor, remove it */
        if (GStextwidth(out_string) > field_wd)
        {
            out_string[strlen(out_string)-1] = '\0';
        }
    }
}

/*---------------------------------------------
add_rect
    Add a rectangle to the list of drawing objects.
---------------------------------------------*/
DrawObj *add_rect(
    int x, int y, int wd, int ht,
    bool fill, bool border,
    GColor f_col, GColor b_col
)
{
    DrawObj *new_object;
    RectObj *rect;

    if ((fill == FALSE) && (border == FALSE))
        return NULL;

    new_object = (DrawObj *) malloc(sizeof(DrawObj));
    if (new_object == NULL) {
        GSealog(0, "Out of memory for new rectangle.\n");
        return NULL;
    }

    new_object->type = rectObj;

    rect = &(new_object->object.rect);
    rect->x = x;
    rect->y = y;
    rect->wd = wd;
    rect->ht = ht;
    rect->fill = fill;
    rect->border = border;
    rect->f_col = f_col;
    rect->b_col = b_col;
```
/* put new object in object list */
if (first_object == NULL)
  first_object = new_object;
else
  last_object->next_object = new_object;
last_object = new_object;
new_object->next_object = NULL;
GServiceEnableMenu("funca_menu", CLEAR_DRAW_MENU_ITEM);

return new_object;

/*-----------------------------------------------*/
add_rect
  Add a text object to the list of drawing objects.
-----------------------------------------------*/
DrawObj *add_text(
  int x, int y,
  char *str,        
  GColor t_col
)
{
  DrawObj *new_object;
  TextObj *text;

  if (strlen(str) == 0)
    return NULL;

  new_object = (DrawObj *) malloc(sizeof(DrawObj));
  if (new_object == NULL) {
    GServiceLog(0, "Out of memory for new text.\n");
    return NULL;
  }

  text = &(new_object->object.text);

  text->str = malloc(strlen(str) + 1);
  if (text->str == NULL) {
    free(new_object);
    GServiceLog(0, "Out of memory for text string %s .\n", str);
    return NULL;
  }

  new_object->type = textObj;

  text->x = x;
  text->y = y;
  text->t_col = t_col;
  strcpy(text->str, str);

  /* put new object in object list */
  if (first_object == NULL)
    first_object = new_object;
  else

```c
last_object->next_object = new_object;
last_object = new_object;
new_object->next_object = NULL;

GSenablemenuitem("funcs_menu", CLEAR_DRAW_MENU_ITEM);

return new_object;
}

/****************************************/
alert_done
Called when "Done" button in PopUpMessage window is pushed.
Removes the PopUpMessage window from the screen.
****************************************/
bool alert_done (  
    void
)
{
    GSremovenamedwindow("alert");
    GSsetcurrentnamedwindow("root");
    show_fill_color(FillColor);
    GSsetcurrentnamedwindow("io");
    AlertShowing = FALSE;
    return TRUE;
}

/****************************************/
char_is_legal
Returns TRUE if ch is a legal input character (A-Z, a-z, '.', '-', or ' ').
Returns FALSE if ch is not a legal input character.
****************************************/
bool char_is_legal (  
    char ch /*--- character to check for legality ---*/
)
{
    return ((isalnum(ch)) || (ch == ' ') || (ch == '-') || (ch == '?'));
}

/****************************************/
clear_draw
Called when "Clear Draw Area" item on "Menu" menu is selected.
Clears the draw area.
****************************************/
bool clear_draw (  
    void
)
{
    int draw_area_x, draw_area_y, draw_area_wd, draw_area_ht;
    GSColor draw_area_color;

    remove_all_objects();

    /*--- draw area is part of "io" window ---*/
    GSsetcurrentnamedwindow("io");
```

/* --- 'clear' draw area by refilling with its background color ---*/
GSgetfieldrect("dragbg", &draw_area_x, &draw_area_y, &draw_area_wd, &draw_area_ht, &draw_area_color);
GSsetcolor(draw_area_color);
GSfillrect(draw_area_x, draw_area_y, draw_area_wd, draw_area_ht);

/* --- disable "Clear Draw Area" menu item, since area is now cleared ---*/
GSdisablemenuitem("funcs_menu", CLEAR_DRAW_MENU_ITEM);
return TRUE;
}

 nationalists
Called when "Clear Text Field" item on "Menu" menu is selected.
Clears the text input field.
--------------------------------------------------------------------------*/
bool clear_text ( void )
{
    int       background_x, background_y, background_wd, background_ht;
    GSColor   background_color;
    int       cursor_x, cursor_y;
    GSColor   cursor_color;
    
    strcpy(TextString, "");
    GSsetcurrentnamedwindow("io");

    /* --- 'clear' field by refilling it with its background color ---*/
    GSgetfieldrect("textbg", &background_x, &background_y, &background_wd, &background_ht, &background_color);
    GSsetcolor(background_color);
    GSfillrect(background_x, background_y, background_wd, background_ht);

    /* --- draw text cursor; use NULL for unneeded wd, ht in GSgetfieldrect ---*/
    GSgetfieldrect("textfg", &cursor_x, &cursor_y, NULL, NULL, &cursor_color);
    GSsetcolor(cursor_color);
    GSwritemsg(cursor_x, cursor_y, "_");

    /* --- disable "Clear Text Area" button, since area is now cleared ---*/
    GSdisablemenuitem("funcs_menu", CLEAR_TEXT_MENU_ITEM);
    return TRUE;
}

/* --- 'do again' ---*/
do_again
    Called when "Again" button in "I2win" window is pressed.
    Writes "Hello World" in field "hello2".
--------------------------------------------------------------------------*/
bool do_again ( void )
{
    write_hello_again();

9.2 Source Code for giltest

/**** erase button, since pressing it again will have no effect *****/
GSinvisbutton("io", "again", DRAW);

GSDisplayhelpmessage(AGAIN_MSG);
return TRUE;
}

/**** do_draw processes drag events in the draw area. */
bool do_draw (int mouse_x, /*--- x-coordinate of mouse cursor ---*/
int mouse_y, /*--- x-coordinate of mouse cursor ---*/
DragStatusType status /*--- type of mouse event ---*/
)
{
int dfg_x, dfg_y;
GSColor dfg_color;
int dbg_x, dbg_y;
int cfg_x, cfg_y;
GSColor cfg_color;
int cbg_x, cbg_y, cbg_wd, cbg_ht;
GSColor cbg_color;
static int initial_x, initial_y;
static int xor_x, xor_y, xor_wd, xor_ht;
BigString coordinate_string;

DrawObj *new_object;

switch (status)
{
case MOUSE_CLICK: /*--- write contents of text area at mouse cursor ---*/
    GSgetfielddirect("dragfg", &dfg_x, &dfg_y, NULL, NULL, &dfg_color);
    new_object = add_text(mouse_x+dfg_x, mouse_y+dfg_y, TextString, dfg_color);
    if (new_object != NULL)
        draw_object(new_object);
    break;

case DRAG_INIT: /*--- initialize XOR box coordinates and dimensions ---*/
    xor_x = initial_x = mouse_x;
    xor_y = initial_y = mouse_y;
    xor_wd = xor_ht = 0;
    break;

case DRAG_PROCESS: /*--- move XOR box ---*/
    GSgetfielddirect("dragbg", &dbg_x, &dbg_y, NULL, NULL, NULL);

    //--- erase old XOR box ---*/
    GSdrawxorbox(dbg_x+xor_x, dbg_y+xor_y, xor_wd, xor_ht);

    //--- compute location and dimensions of new XOR box ---*/
    xor_x = MIN(mouse_x, initial_x);
    xor_y = MIN(mouse_y, initial_y);
    xor_wd = abs(mouse_x - initial_x);
    xor_ht = abs(mouse_y - initial_y);
/*--- draw new XOR box ---*/
GSdrawxorbox(dbg_x+xor_x,dbg_y+xor_y,xor_wd,xor_ht);
break;

case DRAG_FINAL: /*--- erase XOR box; draw a box if parameters are set ---*/
    GSgetfieldrect("dragbg", &dbg_x, &dbg_y, NULL, NULL, NULL);

    /*--- erase XOR box ---*/
    GSdrawxorbox(dbg_x+xor_x, dbg_y+xor_y, xor_wd, xor_ht);
    GSgetfieldrect("dragfg", &dfg_x, &dfg_y, NULL, NULL, &dfg_color);
    new_object = add_rect(dbg_x+xor_x, dbg_y+xor_y, xor_wd, xor_ht, FillBorder, DrawBorder, FillColor, dfg_color);
    if (new_object != NULL)
        draw_object(new_object);
    break;

case MOUSE_INSIDE: /*--- show current mouse cursor coordinates ---*/
    /*--- 'erase' old coordinates ---*/
    GSgetfieldrect("coordbg", &cbg_x, &cbg_y, &cbg_wd, &cbg_ht, &cbg_color);
    GSsetcolor(cbg_color);
    GSfillrect(cbg_x, cbg_y, cbg_wd, cbg_ht);
    /*--- show new coordinates ---*/
    GSgetfieldrect("coordfg", &cfg_x, &cfg_y, NULL, NULL, &cfg_color);
    GSsetcolor(cfg_color);
    sprintf(coordinate_string, "(%d, %d), mouse_x, mouse_y);
    GSwritestr(cfg_x, cfg_y, coordinate_string);
    break;

case MOUSE_OUTSIDE: /*--- erase mouse cursor coordinates ---*/
    GSgetfieldrect("coordbg", &cbg_x, &cbg_y, &cbg_wd, &cbg_ht, &cbg_color);
    GSsetcolor(cbg_color);
    GSfillrect(cbg_x, cbg_y, cbg_wd, cbg_ht);
    break;
}
}

/*--- switch (status) ---*/
return(TRUE);
}

/*----------------------------------------*/
do_intro
    Writes "Hello World" in field "hello!".
    Demonstrates the use and deactivation of independent functions.
    This function is activated in the main function.
----------------------------------------*/

bool do_intro ( void )
{
    int    logo_x, logo_y, logo_wd, logo_ht;
    int    text_x, text_y, text_wd, text_ht;
    GColor  text_color;

    /*--- deactivate this function (it should only ve called once) ---*/
    GSdeactivatefunction("intro");
9.2 Source Code for gltest

    /*--- initialize fill color field to first color of fillcolor list ---*/
    GSsetcurrentnamedwindow("root");
    FillColor = GSgetlistcolor(FILL_COLOR_LIST, INITIAL_ITEM);
    show_fill_color(FillColor);
    GStogglecheck("col_menu", GSgetmenuitemname("col_menu", INITIAL_ITEM));
    GSdisplayversion("version.txt");
    
    write_hello();
    GSdisplayhelpmessage(INTRO_MSG);

    /*--- draw text cursor in text output field ---*/
    GSgetfieldrect("text", &text_x, &text_y, &text_wd, &text_ht, &text_color);
    GSsetcolor(text_color);
    GSwritesmsg(text_x, text_y, ".");
    
    /*--- draw globe logo ---*/
    GSgetfieldrect("logo", &logo_x, &logo_y, &logo_wd, &logo_ht, NULL);
    GSdrawpict(&pict, logo_x, logo_y, logo_wd, logo_ht, 0, 0, 1, 
               logo_converter);
    return TRUE;
}

/*-----------------------------------------------*/

do_quit

Called when "Quit" item on "Menu" menu is selected.
Quits gltest.

-----------------------------------------------*/
bool do_quit (  
void
)
{
    GSquit(NULL);
    return TRUE;
}

/*-----------------------------------------------*/

do_text

Called when a character is typed at the keyboard.
Processes a character typed at the keyboard.

-----------------------------------------------*/
bool do_text (  
int mouse_x, /**** not used here, but required by event handler ****/
int mouse_y, /**** not used here, but required by event handler ****/
char ch /**** character typed at the keyboard ****/
)
{
    int text_x, text_y, text_wd, text_ht;
    GSColor text_color;
    int background_x, background_y, background_wd, background_ht;
    GSColor background_color;
    int length;
    BigString current_window;
    BigString out_string;
    int i;
    
    /*--- Don't process character if alert window is up. ---*/
if (AlertShowing)
{
    GSbell();
    return (FALSE);
}

/*--- Don't process character unless "io" is the current window. ---*/
GSgetcurrentwindowname(current_window);
length = strlen(current_window);
for (i = 0; i < length; i++)
{
    current_window[i] = tolower(current_window[i]);
}
if (strcmp(current_window, "io") != 0)
{
    GSbell();
    return (FALSE);
}

GSsetcurrentnamedwindow("io");
GSgetfieldrect(\"text\", &text_x, &text_y, &text_wd, &text_ht, &text_color);
if (char_is_legal(ch))
{
    length = strlen(TextString);
    /*--- check length of string (in characters) in memory ---*/
    if (length < BIGSTRINGLEN)
    {
        /*--- copy string and add character to temporary buffer ---*/
        strcpy(out_string, TextString);
        out_string[length] = ch;
        out_string[length+1] = '\0';
        /*--- check length of string (in pixels) on screen ---*/
        if (GStextwidth(out_string) <= text_wd)
        {
            /*--- update permanent buffer if string still fits in field ---*/
            strcpy(TextString, out_string);
            if (strlen(TextString) == 1)
            {
                GSendenablemenuitem("funcs_menu", CLEAR_TEXT_MENU_ITEM);
            }
        }
        else /*--- string will not fit in field with new character added ---*/
        {
            /*--- alert user that string would be too long ---*/
            GSalert(-1, TRUE, "typedtoofar", "");
            show_fill_color(GSconvertcolor("mgray"));
            /*--- disable typing until alert window is removed by user ---*/
            AlertShowing = TRUE;
            return FALSE;
        }
    }
}
else if ((ch == '\b') && (strlen(TextString) > 0))
{
    /*--- rubout last character ---*/
    TextString[strlen(TextString)-1] = '\0';
    if (strlen(TextString) == 0) /*--- then there is no more text to clear ---*/
    {

9.2 Source Code for giltest

    GSDisallowItem("funcx_menu", CLEAR_TEXT_MENU_ITEM);
}
strcpy(out_string, TextString);
else
{
    return FALSE;
}
/*--- if we get here, the string has been updated, so rewrite it ---*/
add_cursor_if_space(out_string, text_xd);
GSGetFieldRect("textbg", &background_x, &background_y, &background_wd,
    &background ht, &background color);
GSSetColor(background_color);
GSFillRect(background_x, background_y, background_wd, background ht);
GSSetColor(text_color);
GSSetFont(LARGE);
GSWriteMsg(text_x, text_y, out_string);
return TRUE;
}

/*-----------------------------------------------
draw all objects
  Display all drawing objects.
-----------------------------------------------*/
void draw_all_objects()
{
    DrawObj *object = first_object;
    while (object != NULL) {
        draw_object(object);
        object = object->next_object;
    }
}

/*-----------------------------------------------
draw_object
  Display a drawing object.
-----------------------------------------------*/
void draw_object(
    DrawObj *object
)
{
    TextObj *text;
    RectObj *rect;
    GSGetCurrentNamedWindow("io");
    switch(object->type) {
    case rectObj:
        rect = &(object->object.rect);
        if (rect->fill) {
            GSSetColor(rect->f_col);
        

```
GS\fillrect(rect->x, rect->y, rect->wd, rect->ht);
}
if (rect->border) {
    GSsetcolor(rect->b_col);
    GSdrawrect(rect->x, rect->y, rect->wd, rect->ht);
}
break;

case textObj:
    text = \&(object->object.text);
    GSsetcolor(text->t_col);
    GSwritemsg(text->x, text->y, text->str);
    break;

default:
    GSelog(-1, "Error: unknown object type: %d.\n", object->type);
    break;
}
}

/*-----------------------------------------------
 init_colors
   Convert color names to GSColors for logo.
-----------------------------------------------*/
void init_colors(
    void
)
{
    aqua_color = GSconvertcolor("aqua");
    mgray_color = GSconvertcolor("mgray");
    white_color = GSconvertcolor("white");
}

/*-----------------------------------------------
 init_pict
   Initialize the file for a picture and initialize the picture.
-----------------------------------------------*/
void init_pict(
    GSPicture *pict_ptr, /*--- GIL picture structure ---*/
    FILE *fp, /*--- name of file which contains picture image ---*/
    long file_size;

    void *picture_memory;

    fp = GSopen(pict_file, "rb");
    if (fp == NULL)
    {
        GSelog(-1, "Unable to open logo file %s\n", pict_file);
    }
    file_size = GSfilesize(fp);
    picture_memory = (void *)malloc(file_size-8);
    if (picture_memory == NULL)
    {
        GSelog(-1, "Unable to allocate space for picture in file %s\n", pict_file);
9.2 Source Code for giltest

```c
}
if (!GSinitpic(fp, pict_ptr, picture_memory))
{
    GSeglog(-1, "Unable to initialize picture in file %s\n", pict_file);
}
fclose(fp);

/*---------------------------------------------*
logo_converter
    Converts a pixel code from a runlength encoded image file to a color
    palette index.
---------------------------------------------*/
GSColor logo_converter (unsigned short code, /* pixel code from image run */
    int x,
    int y, /* x, y, z - unused params */
    int z)
{
    switch (code)
    {
    case 255:
        return((unsigned char)-1);
    case 0:
        return(mgray_color);
    case 1:
        return(aqua_color);
    default:
        return(white_color);
    }
}

/*---------------------------------------------*
redraw_all
    Called when the application window is exposed after being covered by
    another window. Redraws the application window.
---------------------------------------------*/
bool redraw_all (void)
{
    ButtonStatusType button_status;
    int  text_x, text_y, text_wd;
    GSColor text_color;
    int  logo_x, logo_y, logo_wd, logo_ht;
    BigString out_string;

    /*--- GSredrawnamedwindow is used to redraw a window which already
        appears on the screen ---*/
    GSredrawnamedwindow("root");
    GSdisplayversion("version.txt");
    show_fill_color(FillColor);
    GSredrawnamedwindow("io");
```
*/--- redraw globe logo ---*/
GSetfieldrect("logo", &logo_x, &logo_y, &logo_wd, &logo_ht, NULL);
GSwdrawpict(&LogOpict, logo_x, logo_y, logo_wd, logo_ht, 0, 0, 1,
    logo_converter);

*/--- redraw text string in text input field ---*/
GSetfieldrect("textfg", &text_x, &text_y, &text_wd, NULL, &text_color);
GSetcolor(text_color);
GSetfont(LARGE);
strcpy(out_string, TextString);
add_cursor_if_space(out_string, text_wd);
GSwritemsg(text_x, text_y, out_string);

*/--- first hello is written in intro, so it is always redrawn ---*/
write_hello();

*/--- second hello is written only after "again" button has been pressed ---*/
GSetbuttonepecs("io", "again", NULL, NULL, NULL, NULL, &button_status);
if (button_status == INVISIBLE)="/*/--- then it has been pressed ---*/
{
    write_hello_again();
}

*/--- redraw the current help message (GIL remembers it for you) ---*/
GDisplayhelpmessage(CURR_MSG);

*/--- redraw alert window if it was showing ---*/
if (AlertShowing)
{
    GAlert(-1, TRUE, "typedtoofar", ");
}
draw_all_objects();

GSetcurrentnamedwindow("io");
return TRUE;
}

/#-----------------------------remove_all_objects-----------------------------#
void remove_all_objects(
    void
)
{
    DrawObj *object = first_object, *next_object;
    while (object != NULL) {
        next_object = object->next_object;
        if (object->type == textObj)
            free(object->object.text.str);
        free(object);
    }
object = next_object;
}

first_object = NULL;
last_object = NULL;
}

/**-----------------------------------------------
 set_color
 Called when a fill color is selected from "Fill Color" menu.
 Sets the fill color to the color listed in position itemnum on the menu.
-----------------------------------------------*/
bool set_color (char *item_name /**< name of selected menu item */)
{
  static int prev_item_num = INITIAL_ITEM;

  int item_num; /**< number of selected menu item
(top item is 0, next lower item is 1, etc.) */
  item_num = GSgetmenuitemnum("col_menu", item_name);

  if selected item is different from old item, GIL turns checkmark
for selected item on, and GStogglecheck here turns old checkmark off;
if selected item is the same as the old item, GIL turns old
checkmark off, and GStogglecheck here turns it back on.

GStogglecheck("col_menu", GSgetmenuitemname("col_menu", prev_item_num));
FillColor = GSgetlistcolor(FILL_COLOR_LIST, item_num);
show_fill_color(FillColor);

/**--- show a different message if color value is unchanged ---*/
if (prev_item_num == item_num)
  GSdisplayhelpmessage(SAME_FILL_COLOR_MSG);
else
  GSdisplayhelpmessage(NEW_FILL_COLOR_MSG);

prev_item_num = item_num;
return TRUE;
}

/**-----------------------------------------------
 show_box_styles
 Called when "Box Style" button is pushed.
 Shows the menu of box styles.
-----------------------------------------------*/
bool show_box_styles (void)
{
  GSdisplayhelpmessage(BOX_STYLE_MENU_MSG);
  return TRUE;
}

/**-----------------------------------------------
show_colors
   Called when "Fill Colors" button in "root" window is pressed.
   Shows the menu of fill colors to choose from.

bool show_colors ()
{
    GString message;
    return TRUE;
}

void show_fill_color (Color fill_color)
{
   int button_x, button_y, button_wd, button_ht;
   int rect_x, rect_y, rect_wd, rect_ht;
   BigString current_window;

   GSgetcurrentwindowname(current_window);
   GSsetcurrentnamedwindow("root");
   GSsetcolor(fill_color);

   /*--- determine area of button available for fill color rectangle ---*/
   GSgetbuttonspecs("root", "color", &button_x, &button_y, &button_wd, &button_ht, NULL);
   GSsetfont(LARGE);
   rect_x = button_x + GSgettextwidth(FILL_COLOR_BUTTON_LABEL) + 10;
   rect_y = button_y + 7;
   rect_wd = button_wd - GSgettextwidth(FILL_COLOR_BUTTON_LABEL) - 15;
   rect_ht = button_ht - 13;

   /*--- draw a solid rectangle of current fill color on "color" button ---*/
   GSfillrect(rect_x, rect_y, rect_wd, rect_ht);

   /*--- if color_is_ "mgray", button is disabled, so make border "mgray" ---*/
   if (fill_color != GSconvertcolor("mgray"))
      GSsetcolor(GSconvertcolor("white"));

   /*--- draw a border around the solid rectangle ---*/
   GSdrawrect(rect_x - 1, rect_y - 1, rect_wd + 2, rect_ht + 2);
   GSsetcurrentnamedwindow(current_window);
}

show_funs_menu
   Called when "Menu" button is pushed.
   Shows the menu of user functions to choose from.

bool show_funs_menu ()


{  
    GSdisplayhelpmessage(MENU_MENU_MSG);  
    return TRUE;  
}

;/*-----------------------------------------*/
toggle_border  
    Called when "Draw Black Border" item on "Box Styles" menu is selected.  
    Toggles the DrawBorder parameter.  
/*-----------------------------------------*/
bool toggle_border(
    char *item_name /*--- name of selected menu item ---*/
)
{
    DrawBorder = !DrawBorder;
    if (DrawBorder)  
    {  
        GSdisplayhelpmessage(BOX_STYLE_CHECK_ON_MSG);  
    }
    else  
    {  
        GSdisplayhelpmessage(BOX_STYLE_CHECK_OFF_MSG);  
    }
    return TRUE;
}

;/*-----------------------------------------*/
toggle_fill  
    Called when "Fill With Color" item on "Box Style" menu is selected.  
    Toggles the FillBorder parameter.  
/*-----------------------------------------*/
bool toggle_fill(
    char *item_name /*--- name of selected menu item ---*/
)
{
    FillBorder = !FillBorder;
    if (FillBorder)  
    {  
        GSdisplayhelpmessage(BOX_STYLE_CHECK_ON_MSG);  
    }
    else  
    {  
        GSdisplayhelpmessage(BOX_STYLE_CHECK_OFF_MSG);  
    }
    return TRUE;
}

;/*-----------------------------------------*/
write_hello  
    write "Hello World." in field "hello1" in "io" window.  
/*-----------------------------------------*/
void write_hello(
    void)
{
    int    hello1_x, hello1_y;
GColor  hello1_color;

    */--- write "Hello World" ---*/
    GsGetfieldrect("hello", &hello1_x, &hello1_y, NULL, NULL, &hello1_color);
    GsSetcolor(hello1_color);
    GsSetFont(LARGE);
    GsSetcurrentnamedwindow("io");
    GsWriteMsg(hello1_x, hello1_y, "Hello World.");
} /*---------------------------------------------------------------*/

    write_hello Again
    Write "Hello World, again." in field "hello2" in "io" window.
    /*----------------------------------------------------------------*/

    void write_hello Again ( void )
{
    int hello2_x, hello2_y;
    GColor  hello2_color;

    /*--- get coordinates and color for message text ---*/
    GsGetfieldrect("hello2", &hello2_x, &hello2_y, NULL, NULL, &hello2_color);

    /*--- set text attributes and write message ---*/
    GsSetcolor(hello2_color);
    GsSetFont(LARGE);
    GsWriteMsg(hello2_x, hello2_y, "Hello World, again.");
} /*----------------------------------------------------------------*/

/*---------------------------------------------------------------*/

FuncNames array
    function name -> function pointer mapping
    /*----------------------------------------------------------------*/

    FuncNames[] = {
    /* string in .inf file          function pointer */
    ---------------          ---------------  */
    { "agn_func",          do_again         },
    { "box_menu",          show_box_styles  },
    { "clear_draw_func",   clear_draw      },
    { "clear_text_func",   clear_text      },
    { "col_menu",          show_colors      },
    { "done_func",         alert_done      },
    { "draw_func",         (bool(*)())do_draw },
    { "funcs_menu",        showfuncs_menu  },
    { "intro",             do_intro        },
    { "kbd_hdlr",          (bool(*)())do_text },
    { "quit_prog_func",    do_quit         },
    { "redraw_func",       redraw_all      },
    { "set_color_func",    set_color       },
    { "toggle_border_func",toggle_border   },
    { "toggle_fill_func",  toggle_fill     },
    };

int NumFuncs = GSnumifuncs(FuncNames);

    /*----------------------------------------------------------------*/

    Main routine: Do initializations and start interface handler.

---

```c
int main (  
    int argc,  
    char **argv
)  
{
    first_object = last_object = NULL;

    /*---- initialize output.log ----*/
    if (!GSinitlog ("vt"))
    {
        printf("Unable to initialize output.log\n");
        GSdelay(TIME_TO_READ_MSG);
        exit(-1);
    }

    init_pict(&LogoPict, GlobeFile);

    /*---- initialize giltest interface ----*/
    GSInterfaceInit("GeoSim Interface Library (GIL) Test",  
        /*---- progrm name ----*/
        "giltest.inf",  
        /*---- interface file name ----*/
        BSTORE,  
        /*---- use backing store ----*/
        "v",  
        /*---- error.log mode ----*/
        G640x480x16,  
        /*---- 16-color graphics mode ----*/
        argc, argv);

    init_colors();

    /*---- activate keyboard input function (do_text) ----*/
    GScharfunc("kbd_hdlr");

    /*---- start event loop ----*/
    GSinterface();
    return(0);
}
```
Index

aliases
  color, 16
buttons
  label alignment, 23
color
  aliases, 16
coordinates
  absolute, 8
GSactivatefunction, 67
GSalert, 56
GSbell, 69
GSbigendian, 69
GScharfunc, 65
GSclearcharfunc, 65
GSclearlastcontrol, 69
GSclick, 69
GSconvertcolor, 67
GScreatenamedpopup, 46
GSdeactivatefunction, 67
GSDelay, 69
GSDisablebutton, 44
GSDisabledragarea, 48
GSDisabledmenuitem, 47
GSDistantlesclist, 55
GSDisplayhelpmessage, 56
GSDisplayversion, 67
GSDoeventsandreturn, 41
GSDohelpscreens, 56
GSDologinscreen, 66
GSDrawinverbox, 62
GSDrawlistitem, 54
GSDrawnamedwindow, 42
GSDrawlistitem, 53
GSDrawxorbox, 62
GSDumpscreen, 68
GSElog, 70
GSENablebutton, 44
GSENabledragarea, 48
GSENabledmenuitem, 47
GSfilesize, 69
GSframedragarea, 51
GSframefield, 50
GSGenreader, 71
GSgetbuttons, 45
GSgetcolor, 60
GSgetcurrentwindowname, 43
GSgetcursormsg, 56
GSgetcursorxy, 67
GSgetdragareaspect, 49
GSgetfieldrect, 50
GSgetfilepictspecs, 65
GSgetfilepixel, 64
GSgetlastcontrol, 69
GSgetlistcolor, 55
GSgetmenutemname, 48
GSgetmenutemnum, 48
GSgetmsg, 60
GSgetpixel, 64
GSgetscrollpos, 55
GSgetwindowspect, 43
GShandledownbutton, 54
GShandlelist, 54
GShandlescllist, 54
GShandleupbutton, 54
GShasFPU, 69
GSinitlog, 70
GSinitpic, 64
GSinterface, 40
GSinterfaceinit, 40
GSinvisiblebutton, 45
GSinvisiblemenuitem, 46
GSLoadwindows, 68
GSmemavail, 67
GSmouseinit, 41
GSmouseoff, 66
GSmove, 66
GSnumfuncs, 69
GSlog, 70
GSopen, 70, 73
GSquit, 41
GSRdboolean, 72
GSrdformat, 73
INDEX

GSrdinteger, 72
GSrdphrase, 72
GSrdreal, 72
GSrdstringio, 72
GSrdword, 72
GSredrawactivewindows, 43
GSredrawnamedwindow, 43
GSremovename, 42
GSremovetopwindow, 43
GSsavewindows, 68
GSclissetup, 52
GSsetactivenamedwindow, 42
GSsetbuttons, 45
GSsetbuttoncontext, 46
GSsetcolor, 60
GSsetcurrentnamedwindow, 42
GSsetdeactivenamedwindow, 43
GSsetdragareas, 49
GSsetfieldrect, 50
GSsetfont, 58
GSsetlinesize, 60
GSsetmenuorigin, 47
GSsetsclistnumitems, 55
GSsetsclistpos, 55
GSsetwindowspecs, 44
GSsettextheight, 58
GSsettextwidth, 58
GSstogglecheck, 47
GSvisiblebutton, 45
GSvisiblemenuitem, 47
GSclearrect, 50
GSdrawellipse, 62
GSdrawfilepict, 63
GSdrawinvertedline, 63
GSdrawline, 61
GSdrawpict, 65
GSdrawpoint, 61
GSdrawrect, 61
GSdrawxorline, 63
GSfastrow, 63
GSfillellipse, 62
GSfillpoly, 61
GSfillrect, 61
GSgetpoint, 62
GSsetmouseposition, 66
GSwriteclippedmsg, 58
GSwritecommanum, 59
GSwriteint, 59
GSwriteitem, 58
GSwritenamedmsg, 60
GSwritereal, 59

independent functions
activating, 28
with GSactivatefunction, 67
deactivating, 28
with GSdeactivatefunction, 67

keyboard handlers
activating
with GScharfunc, 65
with GSdologinscreen, 66
deactivating
with GSclearcharfunc, 65

memory, available
determining amount with GSnumafile, 67
messages
alert, 21
displaying with GSalert, 56

window
alert, 21, 56
current, 8
windows
activating
deactivated by a popup window, 9
with GSsetactivenamedwindow, 42
active, 9
redrawing all, 43
deactivating
with a popup window, 9, 42
with GSsetactivenamedwindow, 43
inactive, 9
popup, 9
restoring status of
with GSloadwindows, 68
saving status of
with GSavewindows, 68