GLUI

A GLUT-Based User Interface Library

by Paul Rademacher

Version 2.0
June 10, 1999

Contents

1 Introduction

1.1 Overview

GLUI is a GLUT-based C++ user interface library which provides controls such as buttons, checkboxes, radio buttons, spinners, and listboxes to OpenGL applications. It is window-system independent, relying on GLUT to handle all system-dependent issues, such as window and mouse management. Features of the GLUI User Interface Library include:

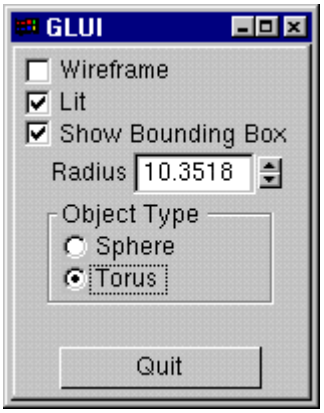
- Complete integration with GLUT toolkit
- Simple creation of a new user interface window with a single line of code
- Support for multiple user interface windows
- Standard user interface controls such as:
  - Buttons
  - Checkboxes for boolean variables
  - Radio Buttons for mutually-exclusive options
  - Editable text boxes for inputting text, integers, and floating-point values
  - Spinners for interactively manipulating integer and floating-point values
  - Arcball controllers for inputting rotation values
  - Translation controllers for inputting X, Y, and Z values
  - Listboxes
  - Static text fields
  - Panels for grouping sets of controls
  - Rollouts (collapsible panels)
  - Separator lines to help visually organize groups of controls
- Controls can generate callbacks when their values change
- Variables can be linked to controls and automatically updated when the value of the control changes (*live variables*)
- Controls can be automatically synchronized to reflect changes in live variables
- Controls can trigger GLUT redisplay events when their values change
- Layout and sizing of controls is automatic
- User can cycle through controls using Tab key

1.2 Background

The OpenGL Utility Toolkit (GLUT) is a popular user interface library for OpenGL applications. It provides a simple interface for handling windows, a mouse, keyboard, and other input devices. It has facilities for nested pop-up menus, and includes utility functions for bitmap and stroke fonts, as well as for drawing primitive graphics objects like spheres, tori, and teapots. Its greatest attraction is its *window system independence*, which (coupled with OpenGL's own window system independence) provides a very attractive environment for developing cross-platform graphics applications.

Many applications can be built using only the standard GLUT input methods - the keyboard, mouse, and pop-up menus. However, as the number of features and options increases, these methods tend to be greatly overworked. It is not uncommon to find glut applications where almost every key on the keyboard is assigned to some function, and where the pop-up menus are large and cumbersome.

The GLUI User Interface Library addresses this problem by providing standard user interface elements such as buttons and checkboxes. The GLUI library is written entirely over GLUT, and contains *no system-dependent code*. A GLUI program will therefore behave the same on SGIs, Windows machines, Macs, or any other system to which GLUT has been ported. Furthermore, GLUI has been designed for programming simplicity, allowing user interface elements to be added with one line of code each.



Sample GLUI window

1.3 What's New in Version 2.0?

GLUI version 2.0 includes the following new features and controls:

- GLUI controls within the main graphics window. This makes GLUI compatible with single-window graphics cards. These GLUI subwindows can be docked to the top, bottom, left, and/or right of the main graphics window, and they can be stacked as well (i.e., multiple subwindows may be docked to the top, left, etc.).
- Functions for cleanly destroying GLUI windows and subwindows.
- Functions for hiding and showing GLUI windows and subwindows.
- `A sync_live_all()`for automatically synchronizing all live variables in all GLUI windows simultaneously.
- Rollouts – collapsible panels for reducing screen clutter.
- Listboxes – allows the user to choose an item from a list of strings.
- Rotation and translation controllers – for easily receiving 3D interaction input from the user.

2 Overview

GLUI is intended to be a simple yet powerful user interface library. This section describes in more detail its main features, including a flexible API, easy and full integration with GLUT, live variables, and callbacks.

2.1 Simple Programming Interface

GLUI has been designed for maximum programming simplicity. New GLUI windows and new controls within them can be created with a single line of code each. GLUI automatically sizes controls and places them within their windows. The programmer does not need to explicitly give X, Y, width, and height parameters for each control - an otherwise cumbersome task.

GLUI provides default values for many parameters in the API. This way, one does not need to place `NULL` or dummy values in the argument list when some feature are not needed. As an example, there are several ways to create a checkbox:

```
GLUI *glui;
 ...
glui->add_checkbox("Click me");
```

Adds a simple checkbox with the name "Click me"

```
glui->add_checkbox("Click me", &state );
```

The variable `state` will now be automatically update to reflect the state of the checkbox (see *live variables* below).

```
glui->add_checkbox( "Click me", &state, 17, callback_fn );
```

Now we have a live variable, *plus* a callback function will be invoked (and passed the value '17') whenever the checkbox changes state.

Note how a default size and position for the checkbox was never specified - GLUI automatically lays out controls in their window.

2.2 Full Integration with GLUT

GLUI is built on top of - and meant to fully interact with - the GLUT toolkit. Existing GLUT applications therefore need very little change in order to use the user interface library (these changes are outlined in Section 2.5 below). Once integrated, the presence of a user interface will be mostly transparent to the GLUT application.

2.3 Live Variables

GLUI can associate *live variables* with most types of controls. These are regular C variables that are automatically updated whenever the user interacts with a GLUI control. For example, a checkbox may have an associated integer variable, to be automatically toggled between one and zero whenever the user checks or unchecks the control. A editable text control may maintain an entire character array as a live variable, such that anything the user types into the text box is automatically copied into the application's character array. This eliminates the need for the programmer to explicitly query each control's state to determine their current value or contents. In addition, a GLUI window can send a *GLUT redisplay message* to another window (i.e., a main graphics window) whenever a value in the interface is changed. This will cause that other window to redraw, automatically using the new values of any live variables. For example, a GLUI window can have a spinner to manipulate the radius of an on-screen object. When the user changes the spinner's value, a live variable (say, `float radius`) is automatically updated, and the main graphics window is sent a redisplay message. The graphics window then redraws itself, using the current (that is, the updated) value of `radius` - unaware that it was changed since the last frame. Live variables help make the GLUI interface transparent to the rest of the application.

Live variables are automatically updated by GLUI whenever the user interacts with a control, but what happens if the user directly changes the value of variable? For example, what if the application changes the radius with a line such as:

```
radius = radius * .05;                  // Updates variable, but not control
```

instead of going through the GLUI API:

```
radius_control->set_float_val( radius * .05 ); // Updates control also
```

Clearly, the first method will leave the variable and the on-screen control out-of-sync. To remedy this, one can *synchronize live variables*. This procedure will check the current value of all live variables in a GLUI window, and compare them with the controls' current values. If a pair does not match (that is, the user changed a live variable without telling GLUI), then the control is automatically updated to reflect the variable. Thus, one can make a series of changes to variables in memory, and then use the single function call `sync_live()` to synchronize the user interface:

```
radius   = radius * .05;          // Make changes to a group of variables that
aperture = aperture + .1;         // are linked to controls
num_segments++;

glui->sync_live();               // Update user interface to reflect these changes
```

If a pointer to a live variable is passed to a control creation function (e.g., `add_checkbox()`), then the current value of that variable will be used as the initial value for the control. Thus, remember to always properly initialize live variables (including strings), before passing them to a control creation function.

2.4 Callbacks

GLUI can also generate callbacks whenever the value of a control changes. Upon creation of a new control, one specifies a function to use as a callback, as well as an integer ID to pass to that function when the control's value

changes. A single function can handle callbacks for multiple controls by using a `switch` statement to interpret the incoming ID value within the callback.

2.5 Usage for standalone GLUI windows

Integrating GLUI with a new or existing GLUT application is very straightforward. The steps are:

1. Add the GLUI library to the link line (e.g., glui32.lib for Windows). The proper order in which to add libraries is: GLUI, GLUT, GLU, OpenGL.

2. #include the file "glui.h" in all sources that will use the GLUI library.

3. Create your regular GLUT windows and popup menus as usual. Make sure to store the window id of your main graphics window, so GLUI windows can later send it redisplay events:

```
int window_id = glutCreateWindow( "Main gfx window" );
```

4. Register your GLUT callbacks as usual (*except the Idle callback, discussed below*).

5. Register your GLUT idle callback (if any) with `GLUI_Master` (a global object which is already declared), to enable GLUI windows to take advantage of idle events without interfering with your application's idle events. If you do not have an idle callback, pass in NULL.

```
GLUI_Master.set_glutIdleFunc( myGlutIdle );
```
or
```
GLUI_Master.set_glutIdleFunc( NULL );
```

6. In your idle callback, explicitly set the current GLUT window before rendering or posting a redisplay event. Otherwise the redisplay may accidently be sent to a GLUI window.

```
void myGlutIdle( void ) {
  glutSetWindow(main_window);
  glutPostRedisplay();
}
```

7. Create a new GLUI window using

```
GLUI *glui = GLUI_Master.create_glui( "name", flags, x, y );
```

   Note that `flags`, `x`, and `y` are optional arguments. If they are not specified, default values will be used. GLUI provides default values for arguments whenever possible.

8. Add controls to the GLUI window. For example, we can add a checkbox and a quit button with:

```
glui->add_checkbox( "Lighting", &lighting );
glui->add_button( "Quit", QUIT_ID, callback_func );
```

9. Let each GLUI window you've created know where its main graphics window is:

```
glui->set_main_gfx_window( window_id );
```

10. Invoke the standard GLUT main event loop, just as in any GLUT application:

```
glutMainLoop();
```

2.6 Usage for GLUI subwindows

Adding GLUI subwindows is slightly more complicated than adding standalone GLUI windows. Since the graphics application and GLUI share window space, a little extra work is required to ensure that they cooperate appropriately.

1. Add the GLUI library to the link line (e.g., glui32.lib for Windows). The proper order in which to add libraries is: GLUI, GLUT, GLU, OpenGL.

2. #include the file "glui.h" in all sources that will use the GLUI library.

3. Create your regular GLUT windows and popup menus as usual. Make sure to store the window id of your main graphics window, so GLUI windows can later send it redisplay events:

```
int main_win = glutCreateWindow( "Main gfx window" );
```

4. Register your GLUT callbacks as usual, *except for the **Keyboard**, **Special**, **Mouse**,**Reshape,** and **Idle** callbacks.* These four must be registered with GLUI (as described below).

5. Register your GLUT Keyboard, Special, and Mouse callbacks, using the following functions:

```
GLUI_Master.set_glutKeyboardFunc( myGlutKeyboard );
GLUI_Master.set_glutSpecialFunc( myGlutSpecial );
GLUI_Master.set_glutMouseFunc( myGlutMouse );
GLUI_Master.set_glutReshapeFunc( myGlutReshape );
```

   Where the "`myGlut`" functions are replaced by the application's respective GLUT callbacks.

6. Register your GLUT idle callback (if any) with GLUI, to enable GLUI windows to take advantage of idle events without interfering with your application's idle events. If you do not have an idle callback, pass in NULL.

```
GLUI_Master.set_glutIdleFunc( myGlutIdle );
```

   or

```
GLUI_Master.set_glutIdleFunc( NULL );
```

7. In your idle callback, explicitly set the current GLUT window before rendering or posting a redisplay event. Otherwise the redisplay may accidently be sent to a GLUI window.

```
void myGlutIdle( void ) {
      glutSetWindow(main_window);
      glutPostRedisplay();
}
```

11. Create a GLUI subwindow using `GLUI_Master::create_glui_subwindow()` For example,

```
GLUI *glui_subwin = GLUI_Master.create_glui_subwindow(main_win,
                        GLUI_SUBWINDOW_RIGHT );
```

    This creates a subwindow inside the GLUT window referenced by `main_win`.

12. Tell the new subwindow which graphics window it should send redisplay events to. For example,

```
glui_subwin->set_main_gfx_window( window_id );
```

13. Add controls to the GLUI subwindow.

14. Repeat steps 11-13 to add more subwindows if needed.

15. For each graphics window that also contains GLUI subwindows, you need to compensate for the subwindows when setting the OpenGL viewports. This is done by adding the following code inside the Reshape callback for the graphics windows:

```
int  tx, ty, tw, th;
GLUI_Master.get_viewport_area( &tx, &ty, &tw, &th );
glViewport( tx, ty, tw, th );
```

The above code needs to be used in place of the standard `glViewport( 0, 0, w, h )` that is found in most applications. As a shortcut to the above code, one can also use the following code:

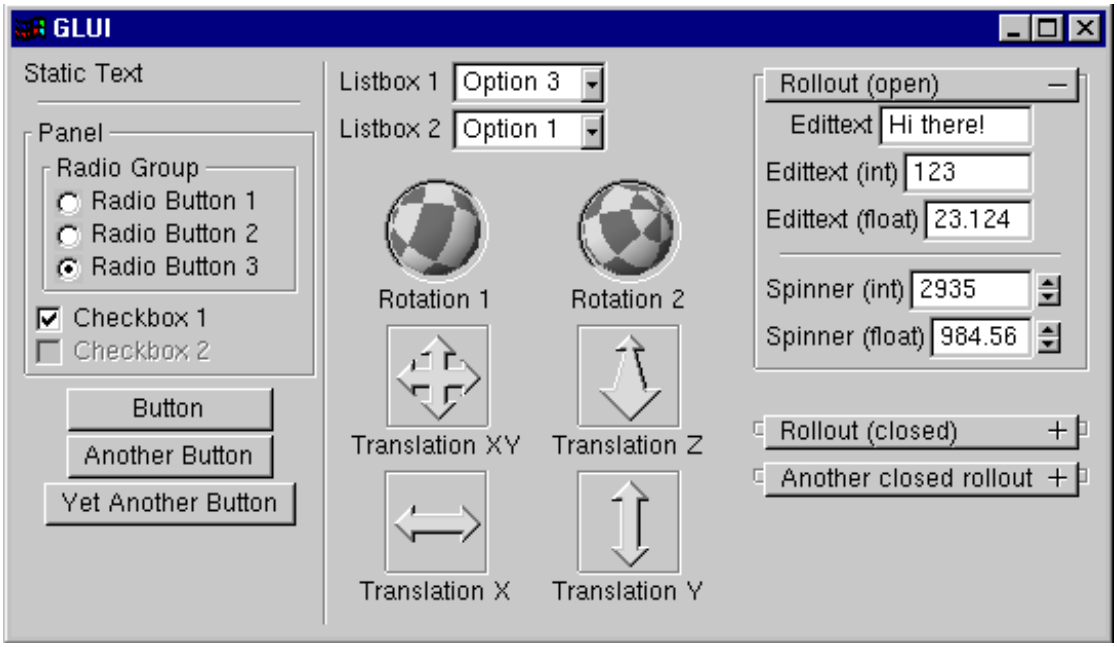
```
GLUI_Master.auto_set_viewport();
```

Which accomplishes the exact same thing.

16. Invoke the standard GLUT main event loop, just as in any GLUT application:

```
glutMainLoop();
```

2.7 List of Controls



Control type	Class name	Used for...	Set/Get values	Live var?	Callback?
Panel	`GLUI_Panel`	grouping controls into boxes	-	-	N
Column	`GLUI_Column`	grouping controls into columns	-	-	N
Rollout	`GLUI_Rollout`	grouping controls into collapsible boxes	-	-	N
Button	`GLUI_Button`	invoking user actions	-	-	Y
Checkbox	`GLUI_Checkbox`	handling booleans	`get_int_val` `set_int_val`	**`int`**	Y
Radio Group, Radio Button	`GLUI_RadioGroup`, `GLUI_RadioButton`	handling mutually-exclusive options	`get_int_val` `set_int_val`	**`int`**	Y
Static Text	`GLUI_StaticText`	plain text labels	`set_text`	-	N
Editable Text	`GLUI_EditText`	text that can be edited – and optionally interpreted as integers or floats. Upper and lower bounds can be placed on integers and floats	`get_int_val` `set_int_val`	**`int`**	Y
			`get_float_val` `set_float_val`	**`float`**	
			`get_text set_text`	**`text`**	
Rotation	`GLUI_Rotation`	Inputting rotation values via an arcball	`get_float_array_val`	**`float [16]`**	Y

Translation	`GLUI_Translation`	Inputting X, Y, and Z values	`get_x`
			`get_y`
			`get_z`
				`float [1] OR [2]`	Y

Listbox	`GLUI_Listbox`	Choosing from a list of items	`get_int_val`	**`int`**	Y
Spinner	`GLUI_Spinner`	interactively manipulating numeric values. Supports single clicks, click-hold, and click-drag. Upper and lower bounds can be specified	`get_int_val` `set_int_val`	**`int`**	Y
			`get_float_val` `set_float_val`	**`float`**	
Separator	`GLUI_Separator`	separating controls with simple horizontal lines	-	-	N

3 Example

```
#include <GL/glut.h>
#include "glui.h"

void myGlutInit();
void myGlutKeyboard(unsigned char Key, int x, int y)
void myGlutMenu( int value )
void myGlutIdle( void )
void myGlutMouse(int button, int button_state, int x, int y )
void myGlutMotion(int x, int y )
void myGlutReshape( int x, int y )
void myGlutDisplay( void );
void control_cb( int ID );

        . . .

void main(int argc, char* argv[])
{
  int   main_window;

  /**  Initialize GLUT and create window - This  **/
  /**  is all regular GLUT code so far           **/

  glutInitDisplayMode( GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH );
  glutInitWindowPosition( 50, 50 );
  glutInitWindowSize( 300, 300 );

  main_window = glutCreateWindow( "GLUI test app" );
  glutKeyboardFunc( myGlutKeyboard );
  glutDisplayFunc( myGlutDisplay );
  glutReshapeFunc( myGlutReshape );
  glutMotionFunc( myGlutMotion );
  glutMouseFunc( myGlutMouse );
  myGlutInit();

  /** Now create a GLUI user interface window and add controls **/

  GLUI *glui = GLUI_Master.create_glui( "GLUI", 0 );
  glui->add_statictext( "Simple GLUI Example" );
  glui->add_separator();
  glui->add_checkbox( "Wireframe", &wireframe, 1, control_cb );
  GLUI_Spinner *segment_spinner =
    glui->add_spinner( "Segments:",GLUI_SPINNER_INT, &segments );
  segment_spinner->set_int_limits( 3, 60, GLUI_LIMIT_WRAP );
  GLUI_EditText *edittext =
     glui->add_edittext( "Text:", GLUI_EDITTEXT_TEXT, text );

  glui->add_column(true);       /**  Begin new column - 'true' indicates   **/
                               /* *  a vertical bar should be drawn        **/

  GLUI_Panel *obj_panel = glui->add_panel( "Object Type" );
  GLUI_RadioGroup *group1 =
     glui->add_radiogroup_to_panel(obj_panel,&obj,3,control_cb);
  glui->add_radiobutton_to_group( group1, "Sphere" );
  glui->add_radiobutton_to_group( group1, "Torus" );
  glui->add_button( "Quit", 0,(GLUI_Update_CB)exit );

  /** Tell GLUI window which other window to recognize as the main gfx window **/
  glui->set_main_gfx_window( main_window );

  /** Register the Idle callback with GLUI (instead of with GLUT)     **/
  GLUI_Master.set_glutIdleFunc( myGlutIdle );

  /** Now call the regular GLUT main loop **/
  glutMainLoop();
}
```

4 API

The GLUI library consists of 3 main classes: `GLUI_Master_Object`, `GLUI`, and `GLUI_Control`. There is a single global `GLUI_Master_Object` object, named `GLUI_Master`. *All GLUI window creation must be done through this object.* This lets the GLUI library track all the windows with a single global object. The `GLUI_Master` is also used to set the GLUT Idle function, and to retrieve the current version of GLUI.

4.1 Windows

This section describes the functions related to window creation and manipulation. The functions listed here belong to two classes: `GLUI_Master_Object` and `GLUI`. Keep in mind that any member function of the `GLUI_Master_Object` class should be invoked from the global object, named `GLUI_Master`, while any function of the `GLUI` class should be invoked via a `GLUI` pointer returned from `GLUI_Master.create_glui()`. For example:

```
float version     = GLUI_Master.get_version();
GLUI *glui_window = GLUI_Master.create_glui( "GLUI" );
glui_window->add_StaticText( "Hello World!" );
```

4.1.1 Initialization

get_version

Returns the current GLUI version.

Usage

```
float   GLUI_Master_Object::get_version( void );
```

Returns: Current GLUI version

create_glui

Creates a new user interface window

Usage

```
GLUI  *GLUI_Master_Object::create_glui( char *name, int flags=0,
                                        int x=-1, int y=-1 );
```

`name` - Name of new GLUI window
`flags` - Initialization flags. No flags are defined in the current version.
`x,y` - Initial location of window. Note that no initial size can be specified, because GLUI automatically resizes windows to fit all controls.

Returns: Pointer to a new GLUI window

create_glui_subwindow

Creates a new user interface subwindow, inside an existing GLUT graphics window.

Usage

```
GLUI  *GLUI_Master_Object::create_glui_subwindow( int window,
                                                  int position );
```

`window` – ID of existing GLUT graphics window

`position` – Position of new subwindow, relative to the GLUT graphics window it is embedded in. This argument can take one of the following values:

```
GLUI_SUBWINDOW_RIGHT
GLUI_SUBWINDOW_LEFT
GLUI_SUBWINDOW_TOP
GLUI_SUBWINDOW_BOTTOM
```

You can place any number of subwindows at the same relative position; in this case, multiple subwindows will simply be stacked on top of one another. For example, if two subwindows are created inside the same GLUT window, and both use `GLUI_SUBWINDOW_TOP`, then the two are placed at the top of the window, although the first subwindow will be above the second.

Returns: Pointer to a new GLUI subwindow

set_glutIdleFunc

Registers a standard GLUT Idle callback `f()` with GLUI. GLUI registers its own Idle callback with GLUT, but calls this user function `f()` after each idle event. Thus every idle event is received by the callback `f()`, but only after GLUI has done its own idle processing. This is mostly transparent to the GLUT application: simply register the idle callback with this function rather than the standard GLUT function `glutIdleFunc()`, and the GLUT application will work as usual. The only caveat is that under the GLUT specification, the current window is undefined in an idle callback. Therefore, your application will need to explicitly set the current window before rendering or posting any GLUT redisplay events:

```
int main_window;

void myGlutIdle( void )
{
   /* ... */

  if ( glutGetWindow() != main_window )
     glutSetWindow(main_window);

   glutPostRedisplay();
}
```

This ensures that the redisplay message is properly sent to the graphics window rather than to a GLUI window.

Usage

```
void   GLUI_Master_Object::set_glutIdleFunc(void (*f)(void));
```

`f` - GLUT Idle event callback function

set_glutReshapeFunc

set_glutKeyboardFunc

set_glutMouseFunc

set_glutSpecialFunc

Registers standard GLUT callbacks with GLUI. GLUI needs to intercept these events for its own processing, but then passes the events to the application. These functions *must be used* instead of the standard GLUT callback registration functions when GLUI subwindows are used. However, these function will also work properly when used with standalone GLUI windows.

Usage

```
void GLUI_Master_Object::set_glutReshapeFunc(
            void (*f)(int width, int height) );

void GLUI_Master_Object::set_glutKeyboardFunc(
            void (*f)(unsigned char key, int x, int y) );

void GLUI_Master_Object::set_glutMouseFunc(
            void (*f)(int, int, int, int) );

void GLUI_Master_Object::set_glutSpecialFunc(
            void (*f)(int key, int x, int y ) );
```

`f` - GLUT event callback function

set_main_gfx_window

Tells a GLUI window which other (standard GLUT) window to consider the main graphics window. When a control in the GLUI window changes value, a redisplay request will be sent to this main graphics window.

Usage

```
void    GLUI::set_main_gfx_window( int window_id );
```

`window_id` - ID of main graphics window. Obtained as the result of `glutCreateWindow()` or with `glutGetWindow()` immediately after the main graphics window is created.

4.1.2 Viewport Management

get_viewport_area

Determines the position and dimensions of the drawable area of the current window. This function is needed when GLUI subwindows are used, since the subwindows will occupy some of the area of a window, which the graphics app should not overwrite. This function should be called within the GLUT reshape callback function. See Section 2.6 for an example of using this function

Usage

```
void    GLUI_Master_Object::get_viewport_area(
            int *x, int *y, int *w, int *h );
```

`x, y, w, h` - When the function returns, these variables will hold the x, y, width, and height of the drawable area of the current window. These values should then be passed into the OpenGL viewport command, `glViewport()`.

auto_set_viewport

Automatically sets the viewport for the current window. This single function is equivalent to the following series of commands:

```
int x, y, w, h;
GLUI_Master.get_viewport_area( &x, &y, &w, &h );
glViewport( x, y, w, h );
```

Usage

```
void    GLUI_Master_Object::auto_set_viewport( void );
```

4.1.3 Window management

get_glut_window_id

Returns the standard GLUT window ID of a GLUI window.

Usage

```
int      GLUI::get_glut_window_id( void );
```

Returns: GLUT window ID of the GLUI window

enable, disable

Enables or disables (grays out) a GLUI window. No controls are active when a GLUI window is disabled.

Usage

```
void     GLUI::enable( void );
void     GLUI::disable( void );
```

hide

Hides a GLUI window or subwindow. A hidden window or subwindow cannot receive any user input.

Usage

```
void     GLUI::hide( void );
```

show

Unhides a previously-hidden GLUI window or subwindow.

Usage

```
void     GLUI::show( void );
```

close

Cleanly destroys a GLUI window or subwindow.

Usage

```
void     GLUI::close( void );
```

close_all

Cleanly destroys *all* GLUI windows and subwindows. This function is called by the following example code:

```
GLUI_Master.close_all();
```

Usage

```
void     GLUI_Master_Object::close_all( void );
```

sync_live

Synchronizes all live variables associated with a GLUI window. That is, it reads the current value of the live variables, and sets the associated controls to reflect these variables. More information on live variables and synchronization, see Section 2.3 above.

Usage

```
void    GLUI::sync_live( void );
```

sync_live_all

Synchronizes every live variable in every GLUI window. This function steps through each GLUI window and subwindow, and invokes its `sync_live()` function. This function is called from the global object, `GLUI_Master`:

```
GLUI_Master.sync_live_all();
```

Usage

```
void    GLUI_Master_Object::sync_live_all( void );
```

4.2 Controls

All controls are derived from the base class `GLUI_Control` As such, they all are created and operated similarly. Section 4.2.1 lists functions that are shared by some or all controls, while the rest of Section 0 lists the specific functions used to create or manage each type of control.

There are two functions to create each type of control. One will be named `add_control()` (where `control` is replaced by the name of the specific control), while the other follows the form `add_control_to_panel()` The second form nests the control within a panel, while the first form places the

control at the top level of the window. Panels are used to group related controls togethers, and panels can be nested within other panels.

Many controls accept *live variables* (Section 2.3) and/or callbacks (Section 2.4). To use live variables (application variables that are automatically updated by the GLUI library), simply pass a pointer to the variable (int, float, or character string) to the `add_control` function as the control is created. To use callbacks, pass in both an integer ID and a callback function. The callback function will be called - with the ID as its single parameter - whenever the control value changes. Multiple controls can share a callback function, which should then use the ID to determine which control invoked it.

Within a callback or at any other time, the current value of a control can be retrieved with one of the functions `get_int_val()`, `get_float_val()`, or `get_text()`, depending on the type of control. For example, a checkbox stores integer values only, while an editable text box may stores a float, an integer, or plain text, depending on what type of text box it is. The values of controls can be set using one of `set_int_val()`, `set_float_val()`, `set_text()`. The documentation for each specific control below list which of these functions it supports.

4.2.1 Common Functions

set_name

Sets the label on a button, checkbox, etc.

Usage

```
void  GLUI_Control::set_name( char *name );
```

`name` - New label for control

set_w, set_h

Sets new minimum width/height for a control. `set_w()` is especially useful to increase the size of the editable text area in an editable text control or spinner.

Usage

```
void  GLUI_Control::set_w( int new_size );
void  GLUI_Control::set_h( int new_size );
```

`new_size` - New minimum width or height for control

get, set

Gets or sets the value of a control. Refer to the individual control descriptions below to see which values can be read and set for each control.

Usage

```
int    GLUI_Control::get_int_val( void );
```

Returns: Current integer value of control

```
float  GLUI_Control::get_float_val( void );
```

Returns: Current floating-point value of control

```
void   GLUI_Control::get_float_array_val( float *float_array_ptr );
```

Returns: There is no return value, but the *array* at `float_array_ptr` will be set with a series of floating-point values. Care must be taken that `float_array_ptr` points to an array of the proper size (e.g., 16 floats for a Rotation control).

```
char  *GLUI_Control::get_text( void );
```
Returns: Pointer to string value of control. Do not modify this string directly - use `set_text()` instead.

```
void   GLUI_Control::set_int_val( int int_val );
void   GLUI_Control::set_float_val( float float_val );
void   GLUI_Control::set_float_array_val( float *float_array_val );
void   GLUI_Control::set_text( char *text);
```

`int_val` -New integer value for control
`float_val` -New floating-point value for control
`float_array_val` -New floating-point *array* values for control
`text` -New text for control. This is the editable text in an editable text box or a spinner, not the label on a button or checkbox - use `set_name()` for that instead.

disable, enable

Disables (grays out) or enables an individual control. A disabled control cannot be activated or used. Disabling a radio group disables all radio buttons within it, and disabling a panel disables all controls within it (including other panels). Enabling behaves similarly.

Usage

```
void   GLUI_Control::enable( void );
void   GLUI_Control::disable( void );
```

set_alignment

Sets the alignment of a control to left-aligned, right-aligned, or centered.

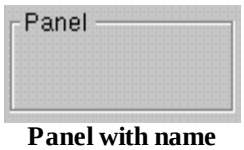
Usage

```
void   GLUI_Control::set_alignment( int align );
```

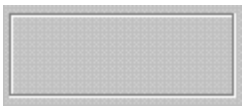
`align` - New alignment. May be one of `GLUI_ALIGN_CENTER`, `GLUI_ALIGN_RIGHT`, or `GLUI_ALIGN_LEFT`.

4.2.2 Panels

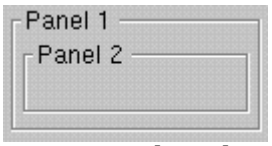
Panels are used to group controls together. An embossed rectangle is drawn around all controls contained within the panel. If the panel is given a name, it will be displayed in the upper-left of the rectangle. Panels may be nested.



Panel with name



Panel without name



Two nested panels

add_panel, add_panel_to_panel

Adds a new panel to a GLUI window, optionally nested within another panel.

Usage

```
GLUI_Panel   *GLUI::add_panel( char *name,
                               int type = GLUI_PANEL_EMBOSSED);

GLUI_Panel   *GLUI::add_panel_to_panel( GLUI_Panel *panel, char *name,
                                        int type = GLUI_PANEL_EMBOSSED);
```

`name` - Label to display in the panel. If string is empty, no label is displayed

`type` - How to draw the panel. The options are:

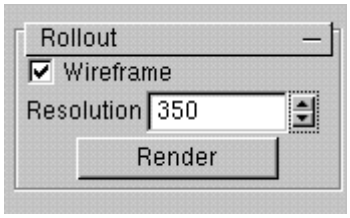
- `GLUI_PANEL_EMBOSSED` - Draw the panel as a sunken box (default)
- `GLUI_PANEL_RAISED` - Draw as a raised box. Name is not displayed.
- `GLUI_PANEL_NONE` - Does not draw a box. Use this for organizing controls into groups without surrounding them with a box.

`panel` - Existing panel to nest new panel in

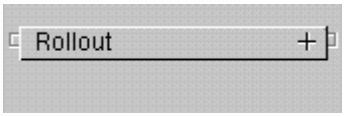
Returns: Pointer to a new panel control

4.2.3 Rollouts

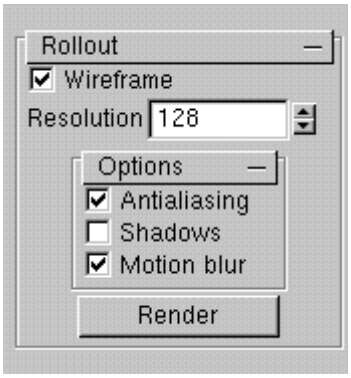
Rollouts are collapsible panels, used to group together related controls. By collapsing, they can greatly reduce on-screen clutter. Rollouts pcan be used interchangeably with Panels. That is, every function of the form `add_control_to_panel()` can accept either a pointer to a Panel or to a Rollout. A rollout can be nested inside another rollout, or inside a panel. Likewise, panels can be nested inside rollouts.



Rollout, expanded



Rollout, collapsed



Nested rollouts

add_rollout, add_rollout_to_panel

Adds a new rollout to a GLUI window, optionally nested within another rollout or panel.

Usage

```
GLUI_Rollout   *GLUI::add_rollout( char *name, int open = true );

GLUI_Rollout   *GLUI::add_rollout_to_panel( GLUI_Panel *panel, char *name,
                                            int open = true );
```

`name` - Label to display in the panel. If string is empty, no label is displayed
`open` - If `true`, rollout will initially be open. If `false`, rollout will initially be collapsed.
`panel` - Panel (or rollout) to place column in.

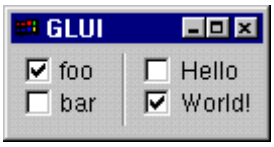
Returns: A pointer to a new Rollout control.

4.2.4 Columns

Controls can be grouped into vertical columns. The function `GLUI::add_column()` begins a new column, and all controls subsequently added will be placed in this new column (until another column is added). Columns can be added within panels, allowing arbitrary layouts to be created.

Examples:

```
glui->add_checkbox("foo");
glui->add_checkbox("bar");
glui->add_column(true);
glui->add_checkbox("Hello");
glui->add_checkbox("World!");
```



```
glui->add_checkbox("foo");
glui->add_checkbox("bar");
GLUI_Panel *panel = glui->add_panel( "Panel" );
glui->add_checkbox_to_panel(panel, "Hello");
glui->add_column_to_panel(panel, true);
glui->add_checkbox_to_panel(panel, "World!");
```

```
glui->add_checkbox( "A" );
glui->add_column( false );
glui->add_checkbox( "B" );
glui->add_column( false );
glui->add_checkbox( "C" );
```

add_column, add_column_to_panel

Begins a new column in a GLUI window, optionally within a panel.

Usage

```
void  GLUI::add_column( int draw_bar = true );
void  GLUI::add_column_to_panel( GLUI_Panel *panel,
                                 int draw_bar = true );
```

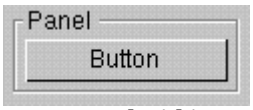
`draw_bar` - If `true`, a vertical bar is drawn at the column boundary.
`panel` - Panel (or rollout) to place column in.

Returns: Pointer to a new button control

4.2.5 Buttons

Buttons are used in conjunction with callbacks to trigger events within an application

Button



Button nested within panel

add_button, add_button_to_panel

Adds a new button to a GLUI window, optionally nested within a panel (or rollout)

Usage

```
GLUI_Button  *GLUI::add_button( char *name, int id=-1,
                                GLUI_Update_CB callback=NULL);

GLUI_Button  *GLUI::add_button_to_panel( GLUI_Panel *panel,
                                  char *name, int id=-1,
                                   GLUI_Update_CB callback=NULL);
```

`name` - Name of button
`id` - If `callback` is defined, it will be passed this integer value
`callback` - Pointer to callback function (taking single `int` argument) to be called when the button is pressed
`panel` - Existing panel (or rollout) to nest button in

Returns: Pointer to a new button control

4.2.6 Checkboxes

Checkboxes are used to handle boolean variables. They take on either the value zero or one. The current value of a checkbox can be read with `GLUI_Checkbox::get_int_val()`, or set with `GLUI_Checkbox::set_int_val()`.

Checkbox

Checkbox nested within panel

add_checkbox, add_checkbox_to_panel

Usage

```
GLUI_Checkbox *GLUI::add_checkbox( char *name,
                                   int *live_var=NULL, int id=-1,
                                    GLUI_Update_CB callback=NULL);

GLUI_Checkbox  *GLUI::add_checkbox_to_panel( GLUI_Panel *panel,
                                    char *name,
                                    int *live_var=NULL, int id=-1,
                                    GLUI_Update_CB callback=NULL);
```

`name` - Name of checkbox

`live_var` - An optional pointer to a variable of type `int`. This variable will be automatically updated with the value of the checkbox (either zero or one) whenever it is toggled.

`id` - If `callback` is defined, it will be passed this integer value

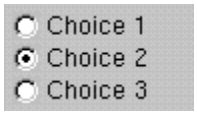
`callback` - Pointer to callback function (taking single `int` argument) to be called when the checkbox state changed is pressed. The callback will be passed the value `id`, listed above

`panel` - Existing panel (or rollout) to nest checkbox in

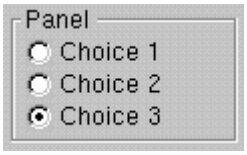
Returns: Pointer to a new checkbox control

4.2.7 Radio Buttons

Radio buttons are used to handle mutually exclusive options. Radio buttons exist only in conjunction with an associated radio group. First a group is created, then buttons are added to it. Radio buttons are assigned a number in the order in which they are added to the group, beginning with zero. The currently selected button can be determined with `GLUI_RadioGroup::get_int_val()`, or set with `GLUI_RadioGroup::set_int_val()`.



Radio group with 3 radio buttons



Radio group with 3 radio buttons, nested within panel

add_radiogroup, add_radiogroup_to_panel

Usage

```
GLUI_RadioGroup *GLUI::add_radiogroup( int *live_var=NULL,
                                       int user_id=-1,
                                        GLUI_Update_CB callback=NULL);

GLUI_RadioGroup *GLUI::add_radiogroup_to_panel(
                                GLUI_Panel *panel,
                               int *live_var=NULL, int user_id=-1,
                                GLUI_Update_CB callback=NULL );
```

`panel` - Panel (or rollout) to nest radio group in
`live_var` - An optional pointer to a variable of type `int`. This variable will be automatically updated with the number of the currently selected radio button. Buttons are numbered from zero in the order in which they are added to the group
`id` - If `callback` is defined, it will be passed this integer value when a new radio button is selected
`callback` - Pointer to callback function (taking single `int` argument) to be called when different radio button is selected. The callback will be passed the value `id`, listed above. Use `GLUI_RadioGroup::get_int_val()` to determine within the callback which button is selected.

Returns: Pointer to a new radio group

add_radiobutton_to_group

Usage

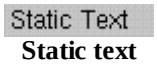
```
GLUI_RadioButton  *GLUI::add_radiobutton_to_group(
                               GLUI_RadioGroup *group, char *name );
```

`group` - Radio group to add button to
`name` - Name for radio button

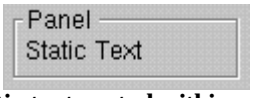
Returns: Pointer to a new radio button

4.2.8 Static Text

Static text controls are used to display simple text labels within a GLUI window. The text to display can be changed with `GLUI_StaticText::set_text()`



Static text



Static text nested within panel

add_statictext, add_statictext_to_panel

Usage

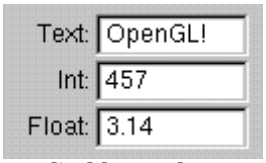
```
GLUI_StaticText  *GLUI::add_statictext( char *name );

GLUI_StaticText  *GLUI::add_statictext_to_panel(
                              GLUI_Panel *panel, char *name );
```

`name` - Text to display
`panel` - Panel (or rollout) to add static text to

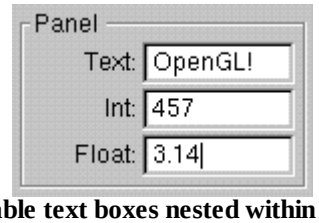
Returns: Pointer to a new static text control

4.2.9 Editable Text Boxes

Editable text boxes can be used to input plain text, integer values, or floating point values. An EditText box designated for integer values will only accept numbers and a preceding minus sign. An EditText box designated for floating-point values will accept numbers, a minus sign, and a decimal point. One can jump ahead or back a word using the Control key in conjunction with the Left or Right keys. Home and End will jump the cursor to the first or last character. EditText controls support text selection using the mouse, or using the Shift key in conjunction with the Left, Right, Control, Home and End keys. The current text of an EditText box can be retrieved using `GLUI_EditText::get_text()`. If the control stores an integer value, it can be retrieved via `GLUI_EditText::get_int_val()`, or a floating-point value using `GLUI_EditText::get_float_val()`. These can also be set using `GLUI_EditText::set_text()`, `GLUI_EditText::set_int_val()`, or `GLUI_EditText::set_float_val()`.



Editable text boxes



Editable text boxes nested within panel

add_edittext, add_edittext_to_panel

Usage

```
GLUI_EditText  *GLUI::add_edittext( char *name,
                                    int data_type=GLUI_EDITTEXT_TEXT,
                                    void *live_var=NULL, int id=-1,
                                      GLUI_Update_CB callback=NULL );

GLUI_EditText  *GLUI::add_edittext_to_panel(
                                       GLUI_Panel *panel, char *name,
                                     int data_type=GLUI_EDITTEXT_TEXT,
                                     void *live_var=NULL, int id=-1,
                                       GLUI_Update_CB callback=NULL );
```

`name` - Label to display left of text box

`data_type` - The type of input the EditText control will accept. The following values are accepted:

- `GLUI_EDITTEXT_TEXT` - The default: regular text input
- `GLUI_EDITTEXT_INT` - Integer input
- `GLUI_EDITTEXT_FLOAT` - Floating-point input

`live_var` - If specified, this must be a pointer to either a character array [of length at least equal to `sizeof(GLUI_String)`], a variable of type `int`, or a variable of type `float`, depending on the value of `data_type`. The string, integer, or float will be modified when the user changes the text in the EditText control

`id` - If `callback` is defined, it will be passed this integer value when the text is changed

`callback` - Pointer to callback function to be called when text is changed. Callback will be passed the single `int` argument '`id`', listed above.

`panel` - Panel (or rollout) to add spinner to

Returns: Pointer to a new editable text control

set_int_limits, set_float_limits

These functions define upper and lower limits on the integer or float values that an editable text box can accept.

Usage

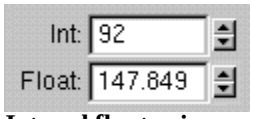
```
void  GLUI_EditText::set_int_limits( int low, int high,
                     int limit_type = GLUI_LIMIT_CLAMP );

void  GLUI_EditText::set_float_limits( float low, float high,
                     int limit_type = GLUI_LIMIT_CLAMP );
```

`low` - Lower bound for acceptable values
`high` - Upper bound for acceptable values
`limit_type` - How to handle out-of-bounds values. If `GLUI_LIMIT_CLAMP`, then out-of-bounds values are simply clamped to the lower or upper limit. If `GLUI_LIMIT_WRAP`, then values that are too low are set to the upper bound, while values that are too high are set to the lower bound. `GLUI_LIMIT_WRAP` is of limited use for editable text boxes, but can be used with spinners to provide continuous cycling over a range (e.g., to continuously increase a rotation amount over the range 0 - 360).

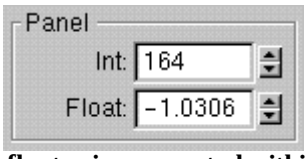
4.2.10 Spinners

A spinner is an integer or floating-point editable text box with two attached arrows, which increase or decrease the current value of the control. The arrows work in three ways: click an arrow once to increase or decrease the spinner's value by a single step, click and hold to continuously increase or decrease the spinner value, or click and drag the mouse to increase and decrease the value as the mouse moves up and down. The rate at which the spinner changes can be varied with the SHIFT and CONTROL keys. Hold SHIFT while initially clicking an arrow to increase the step amount by a factor of 100, or CONTROL to decrease the step amount to $1/100^{th}$ its usual value.

The current value can be retrieved with either `GLUI_Spinner::get_int_val()` or `GLUI_Spinner::get_float_val()`, depending on the type of data stored. It can be set using `GLUI_Spinner::set_int_val()` or `GLUI_EditText::set_float_val()`.



Int and float spinners



Int and float spinners nested within panel

add_spinner, add_spinner_to_panel

Add a new spinner to a GLUI window.

Usage

```
GLUI_Spinner   *GLUI::add_spinner( char *name,
                                   int data_type=GLUI_SPINNER_INT,
                                   void *live_var=NULL, int id=-1,
                                     GLUI_Update_CB callback=NULL );

GLUI_Spinner  *GLUI::add_spinner_to_panel( GLUI_Panel *panel, char *name,
                                        int data_type=GLUI_SPINNER_INT,
                                        void *live_var=NULL, int id=-1,
                                          GLUI_Update_CB callback=NULL );
```

`name` - Label to display

`data_type` - The type of input the Spinner control will accept. The following values are accepted:

- `GLUI_SPINNER_INT` - Integer input
- `GLUI_SPINNER_FLOAT` - Floating-point input

`live_var` - If specified, this must be a pointer to either a variable of type `int` or a variable of type `float`, depending on the value of `data_type`. The integer or float will be modified when the user changes the value

`id` - If `callback` is defined, it will be passed this integer when the spinner's value is modified

`callback` - Pointer to callback function to be called when spinner's value is modified. Callback will be passed the single `int` argument '`id`', listed above.

`panel` - Panel (or rollout) to add spinner to

Returns: Pointer to a new spinner control

set_int_limits, set_float_limits

These functions define upper and lower limits on the integer or float values that an editable text box can accept.

Usage

```
void  GLUI_Spinner::set_int_limits( int low, int high,
                      int limit_type = GLUI_LIMIT_CLAMP );

void  GLUI_Spinner::set_float_limits( float low, float high,
                      int limit_type = GLUI_LIMIT_CLAMP );
```

`low` - Lower bound for acceptable values
`high` - Upper bound for acceptable values
`limit_type` - How to handle out-of-bounds values. If `GLUI_LIMIT_CLAMP`, then out-of-bounds values are simply clamped to the lower or upper limit. If `GLUI_LIMIT_WRAP`, then values that are too low are set to the upper bound, while values that are too high are set to the lower bound. This can be used to provide continuous cycling over a range (e.g., to continuously increase a rotation amount over the range 0 - 360).

set_speed

This function adjusts the rate at which the spinner changes when it is clicked or when the button is held down. This function is used to adjust the spinner responsiveness to either fast or slow machines. That is, for very fast machines, the speed may need to be set to a value less than 1.0, to prevent the spinner from increasing or decreasing too quickly when the button is held down. Likewise, for very slow machines or for applications with a slow update rate, the speed may need to be set to some high value to increase the perceived responsiveness of the interface.

Usage

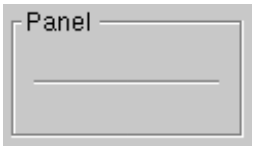
```
void  GLUI_Spinner::set_speed( float speed );
```

`speed` - Rate at which spinner changes. It defaults to `1.0`. Higher values indicate faster change, and low values indicate slower change.

4.2.11 Separators

Separators are simple horizontal lines that can be used to divide a series of controls into groups.

Separator



Separator nested within panel

add_separator, add_separator_to_panel

Adds a separator to a GLUI window

Usage

```
void     GLUI::add_separator( void );

void     GLUI::add_separator_to_panel( GLUI_Panel *panel );
```

`panel` - Panel (or rollout) to add separator to

Returns: -

4.2.12 Rotation Controls

Rotation controls allow the user to input rotations into an application by manipulating an arcball control. The control displays as a checkerboard-textured sphere, which the user manipulates directly. The current rotation of the control is passed to the application as an array of 16 floats, representing a 4×4 rotation matrix. This array can be passed to OpenGL directly to rotate an object, using the function `glMultMatrix()`. Note that because this control deals with pure rotations only (no translation or scaling), the transpose of the 4×4 matrix is the same as its inverse.

The 16 floats can be retrieved from within an application in two ways. The easiest method is to have GLUI set a live array automatically, with an optional callback to the application. That is, the application gives GLUI a 16-float array at the time the rotation control is created. Then, every time the user rotates the arcball, this array is automatically updated. The second way to retrieve the rotation matrix – without using live variables – is by using the function `GLUI_Rotation::get_float_array_val()` (see Section 4.2.1). Likewise, the current rotation of the control can be set using `GLUI_Rotation::set_float_array_val()` which takes as a parameter a pointer to an array of 16 floats.

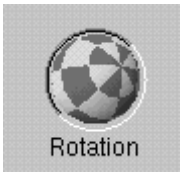
If a 16-element float array is passed to a new Rotation control as a live variable, the control will take its initial rotation from this array (interpreted as a 4×4 matrix). Therefore, the live array needs to be initialized to the identity matrix before being passed to the `GLUI_Rotation::add_rotation()` function:

```
float   array[16] = { 1.0, 0.0, 0.0, 0.0,      0.0, 1.0, 0.0, 0.0,
                      0.0, 0.0, 1.0, 0.0,      0.0, 0.0, 0.0, 1.0  };
```

Alternatively, the control and its associated live array can be initialized to the identity matrix by calling `GLUI_Rotation::reset()` after the rotation control is created.

The rotation control can be constrained to horizontal-only movement by holding the `CTRL` key, or to vertical rotation by holding the `ALT` key.

Rotation controls can optionally keep spinning once the user releases the mouse button. To enable this, use the function `GLUI_Rotation::set_spin()`. Note that spinning should be disabled in performance-critical applications (it is disabled by default).



Rotation control

add_rotation, add_rotation_to_panel

Add a new rotation control to a GLUI window.

Usage

```
GLUI_Rotation   *GLUI::add_rotation( char *name,
                                     float *live_var=NULL, int id=-1,
                                       GLUI_Update_CB callback=NULL );

GLUI_Rotation   *GLUI::add_rotation_to_panel( GLUI_Panel *panel,
                                     char *name,
                                     float *live_var=NULL, int id=-1,
                                       GLUI_Update_CB callback=NULL );
```

`name` - Label to display

`live_var` - If non-null, this must be a pointer to a `float` array of size 16. ***NOTE:** If an array smaller than 16 floats is specified, the application will crash*. This array is treated internally as a 4×4 rotation matrix.

`id` - If `callback` is non-null, then the callback function will be passed this integer id whenever the user rotates the control

`callback` - If non-null, must point to a function with a single `int` parameter and no return value. This function will be called (and passed the single value `id` given above) whenever the control is rotated.

`panel` - An existing GLUI panel (or rollout) to add the control to.

Returns: A pointer to a new GLUI_Rotation control

get_float_array_val, set_float_array_val

Get/set the current rotation.

Usage

```
void   GLUI_Rotation::get_float_array_val( float *array_ptr );

void   GLUI_Rotation::set_float_array_val( float *array_ptr );
```

`array_ptr` - Pointer to a 16-element array of type `float`.

Returns: –

set_spin

Set the damping factor when the arcball is spinning. Higher values indicate less damping. By passing in the maximum value of `1.0`, the arcball will exhibit no damping, and will rotate indefinitely once spun by the user. By setting the value to `0.0`, spinning is disabled altogether. Typical values for the damping factor are `.95` – `.99`. Note that spinning is disabled by default, since it may slow down performance-critical applications.

Usage

```
void   GLUI_Rotation::set_spin( float damping_factor );
```

`damping_factor` - Floating-point value between zero and one. Zero disables spinning, while one makes the arcball spin indefinitely with no slowdown.

Returns: –

reset

Resets the rotation of the control (sets the rotation matrix to the identity matrix). The damping factor (see `set_spin` above) is not changed by this function.

Usage

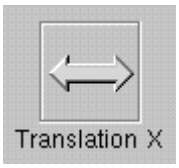
```
void   GLUI_Rotation::reset( void );
```

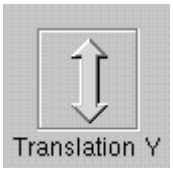
Returns: –

4.2.13 Translation Controls

Translation controls allow the user to manipulate X, Y, and Z values for 3D objects by clicking on on-screen arrows. The rate of change of translation can be varied by holding down SHIFT for fast movement (100 times faster), or CTRL for slow movement (100 times slower). The function `GLUI_Translation::set_scaling()` can be used to set an overall movement scaling factor.

The four types of translation controls are shown below. When using the XY control (on the left), movement can be restricted to a single axis (either X or Y) by holding down ALT and clicking on either the horizontal or the vertical arrows.





add_translation, add_translation_to_panel

Add a new translation control to a GLUI window.

Usage

```
GLUI_Translation  *GLUI::add_translation( char *name, int trans_type,
                                          float *live_var=NULL, int id=-1,
                                            GLUI_Update_CB callback=NULL );

GLUI_Translation  *GLUI::add_translation_to_panel( GLUI_Panel *panel,
                                          char *name, int trans_type,
                                          float *live_var=NULL, int id=-1,
                                            GLUI_Update_CB callback=NULL );
```

`name` - Label to display

`trans_type` - Specifies the type of translation that this control should provide. Choices are:

- `GLUI_TRANSLATION_XY` – Provides X and Y translation
- `GLUI_TRANSLATION_X` – Translation in X only
- `GLUI_TRANSLATION_Y` – Translation in Y only
- `GLUI_TRANSLATION_Z` – Translation in Z only

`live_var` - If non-null, this must be a pointer to a `float` array. The size of this array will depend on the translation type. If the translation type is `XY`, then the array must be of size two (the first element will correspond to the X position, and the second to the Y position). If the translation is not `XY`, then the array must be of size 1 (a single-element array). This element will correspond to either an X, Y, or Z position – depending on the type of the translation control.

`id` - If `callback` is non-null, then the callback function will be passed this integer id whenever the user translates the control.

`callback` - If non-null, must point to a function with a single `int` parameter and no return value. This function will be called (and passed the single value `id` given above) whenever the control is translated.

`panel` - An existing GLUI panel (or rollout) to add the control to.

Returns: A new Translation control

get_x, get_y, get_z

set_x, set_y, set_z

The X, Y, or Z values of a particular translation control can be read or written directly by using these special `get` and `set` functions. These should be used instead of the standard `get_float_val()` and `set_float_val()` functions that are used for other types of controls. Note that `get_x()` should only be used with either a `GLUI_TRANSLATION_X` or a `GLUI_TRANSLATION_XY` control, and similarly for `get_y()`. The `get_z()` function should only be used with a `GLUI_TRANSLATION_Z` control. The same applies for the `set` functions.

Usage

```
float   GLUI_Translation::get_x( void );
float   GLUI_Translation::get_y( void );
float   GLUI_Translation::get_z( void );

void    GLUI_Translation::set_x( float x );
void    GLUI_Translation::set_y( float y );
void    GLUI_Translation::set_z( float z );
```

set_speed

This function determines how fast a translation control changes in response to user mouse movement. All translations reported to the user are first multiplied by this speed factor before being passed on to the application. Applications that need translations to vary slowly should set the speed to a small number below 1.0. Applications that need large translations should set this scaling factor to some large number. The speed value defaults to 1.0.

Usage

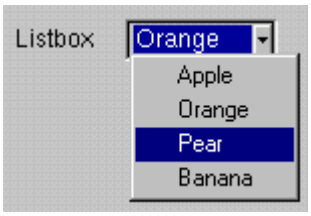
```
void  GLUI_Translation::set_speed( float speed_factor );
```

`speed_factor` - Value to multiply all translations by before passing them on to the application

4.2.14 Listboxes

Listbox controls allow the user to choose from a set of text options. When the user clicks on a listbox, a list of text entries drops down, from which the user selects one. The currently-selected text entry is then displayed in the listbox.

Listbox



Listbox, selected

Each text entry in a listbox is associated with a numerical ID. This ID is explicitly assigned when the text entry is added to the listbox. The currently-selected entry can be determined with `GLUI_Listbox::get_int_val()`, or set with `GLUI_Listbox::set_int_val()`.

add_listbox, add_listbox_to_panel

Add a new listbox to a GLUI window.

Usage

```
GLUI_Listbox   *GLUI::add_listbox( char *name,
                                   void *live_var=NULL, int id=-1,
                                    GLUI_Update_CB callback=NULL );

GLUI_Listbox   *GLUI::add_listbox_to_panel( GLUI_Panel *panel,
                                   char *name,
                                   void *live_var=NULL, int id=-1,
                                    GLUI_Update_CB callback=NULL );
```

`name` - Label to display

`live_var` - An optional pointer to a variable of type `int`. This variable will be automatically updated with the numerical ID of the currently-selected text entry within the listbox.

`id` - ID for this listbox. If `callback` is defined, it will be passed this integer value when a new listbox entry is selected.

`callback` - Pointer to callback function (taking single `int` argument) to be called when a different entry is selected. The callback will be passed the value `id`, listed above. Use `GLUI_Listbox::get_int_val()` to determine within the callback which entry was selected.

`panel` - An existing GLUI panel (or rollout) to add the control to.

Returns: Pointer to new Listbox control

add_item

Add a new entry to an existing Listbox.

Usage

```
int          GLUI_Listbox::add_item( int id, char *text );
```

`id` - Numerical ID for this entry. This is the integer value that is returned by the function `GLUI_Listbox::get_int_val()`, or set with `GLUI_Listbox::set_int_val()`.

`text` - Text label for this entry. This string must be less than 300 characters long.

Returns: `true` if the entry was added to the listbox. Otherwise, `false`.

delete_item

Delete an entry from an existing Listbox. The entry can be referenced by either its numerical ID or by its text.

Usage

```
int          GLUI_Listbox::delete_item( int id );

int          GLUI_Listbox::delete_item( char *text );
```

`id` - Numerical ID for this entry. This is the integer value that is returned by the function `GLUI_Listbox::get_int_val()`, or set with `GLUI_Listbox::set_int_val()`

`text` - Text label for this entry. This string must be less than 300 characters long.

Returns: `true` if the entry was found in the listbox, and succesfully removed. Otherwise, `false`.

5 Usage Advice

- Register your Idle callback with GLUI, not with GLUT. Otherwise, GLUI will be unable to receive idle events (*do not register your idle callback with both GLUI and GLUT, since one will override the other).* Also, check the current window in your Idle callback before rendering or posting a GLUT redisplay event, and set it to the main graphics window if necessary, since GLUT does not guarantee which will be the current window when an Idle callback function is called.

- If you do not have an Idle callback, pass `NULL` to `GLUI_Master.set_glutIdleFunc()`

- If live variables are used, GLUI will take their value as the initial value for a control. Always initialize your live variables to some appropriate value before passing them to GLUI. For example, the following code may initialize a checkbox to an invalid value (neither one or zero):

```
int some_var;  // This variable may contain any integer value
glui->add_checkbox( "Some Var", &some_var );
```

  Instead, the code should be written as:

```
int some_var = 1;  // Or zero, if appropriate
glui->add_checkbox( "Some Var", &some_var );
```

- String buffers passed to editable text controls must be at least of size `sizeof(GLUI_String)` Otherwise, a segmentation fault may occur as GLUI overwrites the buffer.

- Do not pass in a static string as a live variable in an editable text control, as GLUI will attempt to use that space as a buffer for user-typed text:

```
glui->add_edittext( "Text", GLUI_EDITTEXT_TEXT, "Hello!" );
            // This is incorrect, as "Hello!" is a static string

char   buffer[sizeof(GLUI_String)];
glui->add_edittext( "Text", GLUI_EDITTEXT_TEXT, buffer );
            // This will work without crashing

GLUI_String   buffer2;
glui->add_edittext( "Text", GLUI_EDITTEXT_TEXT, buffer2 );
            // This will also work
```

- Also, do not pass in a pointer to a *local* variable as a live variable, as this pointer will be invalid outside the local function, and a segmentation fault will likely occur.

- The various set/get functions (`set_int_val, set_float_val, set_text, get_...`) alter the current value of a control, and *also* update any associated live variables. They will not generate a callback.

- Remember to call `GLUI::set_main_gfx_window()` to link a standard GLUT window (typically the main graphics window) to each GLUI window. This allows GLUI to generate a redisplay event for the graphics window whenever a control value changes.

- During a GLUI callback, the current GLUT window is set to the main graphics window, provided one has been defined with `GLUI::set_main_gfx_window()` If none has been defined, then the current window is undefined - be sure to explicitly set it to the desired window before executing any OpenGL functions.