The MUI (Micro User Interface) facility

Prerequisites

An understanding of event-driven programming and some experience using the simple events and callbacks from the GLUT toolkit in OpenGL, and some review of interface capabilities in standard applications.

Introduction

There are many kinds of interface tools that we are used to seeing in applications but that we cannot readily code in OpenGL, even with the GLUT toolkit. Some of these are provided by the MUI facility that is a universal extension of GLUT for OpenGL. With MUI you can use sliders, buttons, text boxes, and other tools that may be more natural for many applications than the standard GLUT capabilities. Of course, you may choose to write your own tools as well, but you may choose to use your time on the problem at hand instead of writing an interface, so the MUI tools may be just what you want.

MUI has a good bit of the look and feel of the X-Motif interface, so do not expect applications you write with this to look like they are from either the Windows or Macintosh world. Instead, focus on the functionality you need your application to have, and find a way to get this functionality from the MUI tools. The visible representation of these tools are called *widgets*, just as they are in the X Window System, so you will see this term throughout thest notes.

This chapter is built on Steve Baker's "A Brief MUI User Guide," and it shares similar properties: it is based on a small number of examples and some modest experimental work. It is intended as a guide, not as a manual, though it is hoped that it will contribute to the literature on this useful tool.

Definitions

The capabilities of MUI include pulldown menus, buttons, radio buttons, text labels, text boxes, and vertical and horizontal sliders. We will outline how each of these work below and will include some general code to show how each is invoked.

The main thing you must realize in working with MUI is that MUI takes over the event handling from GLUT, so you cannot mix MUI and GLUT event-handling capabilities in the same window. This means that you will have to create separate windows for your MUI controls and for your display, which can feel somewhat clumsy. This is a tradeoff you must make when you design your application — are you willing to create a different kind of interface than you might expect in a traditional application in order to use the extra MUI functionality? Only you can say. But before you can make that choice, you need to know what each of the MUI facilities can do.

Menu bars: A MUI menu bar is essentially a GLUT menu that is bound to a MUI object and then that object is added to a UIlist. Assuming you have defined an array of GLUT menus named myMenus[...], you can use the function to create a new pulldown menu and then use the function to add new menus to the pulldown menu list:

```
muiObject *muiNewPulldown();
muiAddPulldownEntry(muiObject *obj, char *title, int glut_menu, int is_help)
```

An example of the latter function would be

```
myMenubar = muiNewPulldown();
muiAddPulldownEntry(myMenubar, "File", myMenu, 0);
```

where the is_help value would be 1 for the last menu in the menu bar, because traditionally the help menu is the rightmost menu in a menu bar.

According to Baker, there is apparently a problem with the pulldown menus when the GLUT window is moved or resized. The reader is cautioned to be careful in handling windows when the MUI facility is being used.

Buttons: a button is presented as a rectangular region which, when pressed, sets a value or carries out a particular operation. Whenever the cursor is in the region, the button is highlighted to show that it is then selectable. A button is created by the function

```
muiNewButton(int xmin, int xmax, int ymin, int ymax)
```

that has a `muiObject *` return value. The parameters define the rectangle for the button and are defined in window (pixel) coordinates, with (0,0) at the lower left corner of the window.

Radio buttons: radio buttons are similar to standard buttons, but they come in only two fixed sizes (either a standard size or a mini size). The buttons can be designed so that more than one can be pressed (to allow a user to select any subset of a set of options) or they can be linked so that when one is pressed, all the others are un-pressed (to allow a user to select only one of a set of options). Like regular buttons, they are highlighted when the cursor is scrolled over them.

You create radio buttons with the functions

```
muiObject *muiNewRadioButton(int xmin, int ymin)
muiObject *muiNewTinyRadioButton(int xmin, int ymin)
```

where the `xmin` and `ymin` are the window coordinates of the lower left corner of the button. The buttons are linked with the function

```
void muiLinkButtons(button1, button2)
```

where `button1` and `button2` are the names of the button objects; to link more buttons, call the function with overlapping pairs of button names as shown in the example below. In order to clear all the buttons in a group, call the function below with any of the buttons as a parameter:

```
void muiClearRadio(muiObject *button)
```

Text boxes: a text box is a facility to allow a user to enter text to the program. The text can then be used in any way the application wishes. The text box has some limitations; for example, you cannot enter a string longer than the text box's length. However, it also gives your user the ability to enter text and use backspace or delete to correct errors. A text box is created with the function

```
muiObject *muiNewTextbox(xmin, xmax, ymin)
```

whose parameters are window coordinates, and there are functions to set the string:

```
muiSetTBString(obj, string)
```

to clear the string:

```
muiClearTBString(obj)
```

and to get the value of the string:

```
char *muiGetTBString (muiObject *obj).
```

Horizontal sliders: in general, sliders are widgets that return a single value when they are used. The value is between zero and one, and you must manipulate that value into whatever range your application needs. A slider is created by the function

```
muiNewHSlider(int xmin,int ymin,int xmax,int scenter,int shalf)
```

where `xmin` and `ymin` are the screen coordinates of the lower left corner of the slider, `xmax` is the screen coordinate of the right-hand side of the slider, `scenter` is the screen coordinate of the center of the slider's middle bar, and `shalf` is the half-size of the middle bar itself. In the event callback for the slider, the function `muiGetHSVal(muiObject *obj)` is used to return the value (as a float) from the slider to be used in the application. In order to reverse the process — to make the slider represent a particular value, use the `muiSetHSVal(muiObject *obj)` function.

Vertical sliders: vertical sliders have the same functionality as horizontal sliders, but they are aligned vertically in the control window instead of horizontally. They are managed by functions that are almost identical to those of horizontal sliders:

```
muiNewVSlider(int xmin,int ymin,int ymax,int scenter,int shalf)
muiGetVSVal(muiObject *obj)
muiSetVSVal(muiObject *obj)
```

Text labels: a text label is a piece of text on the MUI control window. This allows the program to communicate with the user, and can be either a fixed or variable string. To set a fixed string, use

```
muiNewLabel(int xmin, int ymin, string)
```

with xmin and ymin setting the lower left corner of the space where the string will be displayed. To define a variable string, you give the string a `muiObject` name via the variation

```
muiObject *muiNewLabel(int xmin, int ymin, string)
```

to attach a name to the label, and use the `muiChangeLabel(muiObject *, string)` function to change the value of the string in the label.

Using the MUI functionality

MUI widgets are managed in UI lists. You create a UI list with the `muiNewUIList(int)` function, giving it an integer name with the parameter, and add widgets to it as you wish with the function `muiAddToUIList(listid, object)`. You may create multiple lists and can choose which list will be active, allowing you to make your interface context sensitive. However, UI lists are essentially static, not dynamic, because you cannot remove items from a list or delete a list.

All MUI capabilities can be made visible or invisible, active or inactive, or enabled or disabled. This adds some flexibility to your program by letting you customize the interface based on a particular context in the program. The functions for this are:

```
void muiSetVisible(muiObject *obj, int state);
void muiSetActive(muiObject *obj, int state);
void muiSetEnable(muiObject *obj, int state);
int muiGetVisible(muiObject *obj);
int muiGetActive(muiObject *obj);
int muiGetEnable(muiObject *obj);
```

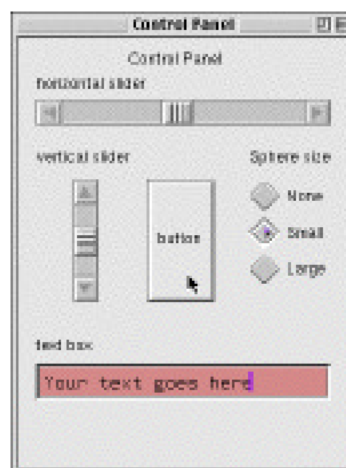

Figure: the set of MUI facilities on a single window

The figure above shows most of the MUI capabilities: labels, horizontal and vertical sliders, regular and radio buttons (one radio button is selected and the button is highlighted by the cursor as

shown), and a text box. Some text has been written into the text box. This gives you an idea of what the standard MUI widgets look like, but because the MUI source is available, you have the opportunity to customize the widgets if you want — though this is beyond the scope of this discussion.

Layout is facilitated by the ability to get the size of a MUI object with the function

```
void muiGetObjectSize(muiObject *obj, int *xmin, int *ymin,
                      int *xmax, int *ymax);
```

MUI object callbacks are optional (you would probably not want to register a callback for a fixed text string, for example, but you would with an active item such as a button). In order to register a callback, you must name the object when it is created and must link that object to its callback function with

```
void muiSetCallback(muiObject *obj, callbackFn)
```

where a callback function has the structure

```
void callbackFn(muiObject *obj, enum muiReturnValue)
```

Note that this callback function need not be unique to the object; in the example below we define a single callback function that is registered for three different sliders and another to handle three different radio buttons, because the action we need from each is the same; when we need to know which object handled the event, this information is available to us as the first parameter of the callback.

If you want to work with the callback return value, the declaration of the `muiReturnValue` is:

```
enum muiReturnValue {
      MUI_NO_ACTION,
      MUI_SLIDER_MOVE,
      MUI_SLIDER_RETURN,
      MUI_SLIDER_SCROLLDOWN,
      MUI_SLIDER_SCROLLUP,
      MUI_SLIDER_THUMB,
      MUI_BUTTON_PRESS,
      MUI_TEXTBOX_RETURN,
      MUI_TEXTLIST_RETURN,
      MUI_TEXTLIST_RETURN_CONFIRM
};
```

so you can look at these values explicitly. For the example below, the button press is assumed because it is the only return value associated with a button, and the slider is queried for its value instead of handling the actual MUI action.

Some examples

Let’s consider a simple application and see how we can create the controls for it using the MUI facility. The application is color choice, commonly handled with three sliders (for R/G/B) or four sliders (for R/G/B/A) depending on the need of the user. This application typically provides a way to display the color that is chosen in a region large enough to reduce the interference of nearby colors in perceiving the chosen color. The application we have in mind is a variant on this that not only shows the color but also shows the three fixed-component planes in the RGB cube and draws a sphere of the selected color (with lighting) in the cube.

The design of this application is built on the project in the “Science Projects” section that shows three cross-sections of a real function of three variables. However, instead of keyboard control for the cross-sections, we use a control built on MUI sliders. We also add radio buttons to allow the user to define the size of the sphere.

Selected code for this application includes declarations of muiObjects, callback functions for sliders and buttons, and the code in the main program that defines the MUI objects for the program, links them to their callback functions, and adds them to the single MUI list we identify. The main issue is that MUI callbacks, like the GLUT callbacks we met earlier, have few parameters and do most of their work by modifying global variables that are used in the other modeling and rendering operations.

```
// selected declarations of muiObjects and window identifiers
muiObject *Rslider, *Gslider, *Bslider;
muiObject *Rlabel, *Glabel, *Blabel;
muiObject *noSphereB, *smallSphereB, *largeSphereB;
int muiWin, glWin;

// callbacks for buttons and sliders
void readButton(muiObject *obj, enum muiReturnValue rv) {
   if ( obj == noSphereB )
      sphereControl = 0;
   if ( obj == smallSphereB )
      sphereControl = 1;
   if (obj == largeSphereB )
      sphereControl = 2;
  glutSetWindow( glWin );
  glutPostRedisplay();
}

void readSliders(muiObject *obj, enum muiReturnValue rv) {
   char rs[32], gs[32], bs[32];
   glutPostRedisplay();

   rr = muiGetHSVal(Rslider);
   gg = muiGetHSVal(Gslider);
   bb = muiGetHSVal(Bslider);

   sprintf(rs,"%6.2f",rr);
   muiChangeLabel(Rlabel, rs);
   sprintf(gs,"%6.2f",gg);
   muiChangeLabel(Glabel, gs);
   sprintf(bs,"%6.2f",bb);
   muiChangeLabel(Blabel, bs);

   DX = -4.0 + rr*8.0;
   DY = -4.0 + gg*8.0;
   DZ = -4.0 + bb*8.0;

   glutSetWindow(glWin);
   glutPostRedisplay();
}

void main(int argc, char** argv)
{
   char rs[32], gs[32], bs[32];
// Create MUI control window and its callbacks
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA);
   glutInitWindowSize(270,350);
   glutInitWindowPosition(600,70);
   muiWin = glutCreateWindow("Control Panel");
   glutSetWindow(muiWin ;
   muiInit();
```

```
    muiNewUIList(1);
    muiSetActiveUIList(1);

// Define color control sliders
    muiNewLabel(90, 330, "Color controls");

    muiNewLabel(5, 310, "Red");
    sprintf(rs,"%6.2f",rr);
    Rlabel = muiNewLabel(35, 310, rs);
    Rslider = muiNewHSlider(5, 280, 265, 130, 10);
    muiSetCallback(Rslider, readSliders);

    muiNewLabel(5, 255, "Green");
    sprintf(gs,"%6.2f",gg);
    Glabel = muiNewLabel(35, 255, gs);
    Gslider = muiNewHSlider(5, 225, 265, 130, 10);
    muiSetCallback(Gslider, readSliders);

    muiNewLabel(5, 205, "Blue");
    sprintf(bs,"%6.2f",bb);
    Blabel = muiNewLabel(35, 205, bs);
    Bslider = muiNewHSlider(5, 175, 265, 130, 10);
    muiSetCallback(Bslider, readSliders);

// define radio buttons
    muiNewLabel(100, 150, "Sphere size");
    noSphereB    = muiNewRadioButton(10, 110);
    smallSphereB = muiNewRadioButton(100, 110);
    largeSphereB = muiNewRadioButton(190, 110);
    muiLinkButtons(noSphereB, smallSphereB);
    muiLinkButtons(smallSphereB, largeSphereB);
    muiLoadButton(noSphereB,    "None");
    muiLoadButton(smallSphereB, "Small");
    muiLoadButton(largeSphereB, "Large");
    muiSetCallback(noSphereB,    readButton);
    muiSetCallback(smallSphereB, readButton);
    muiSetCallback(largeSphereB, readButton);
    muiClearRadio(noSphereB);

// add sliders and radio buttons to UI list 1
    muiAddToUIList(1, Rslider);
    muiAddToUIList(1, Gslider);
    muiAddToUIList(1, Bslider);
    muiAddToUIList(1, noSphereB);
    muiAddToUIList(1, smallSphereB);
    muiAddToUIList(1, largeSphereB);

// Create display window and its callbacks
    ...
}
```

The presentation and communication for this application are shown in the figure below. As the sliders set the R, G, and B values for the color, the numerical values are shown above the sliders and the three planes of constant R, G, and B are shown in the RGB cube. At the intersection of the three planes is drawn a sphere of the selected color in the size indicated by the radio buttons. The RGB cube itself can be rotated by the usual keyboard controls so the user can compare the selected color with nearby colors in those planes, but you have the usual issues of active windows:

you must make the display window active to rotate the cube, but you must make the control window active to use the controls.

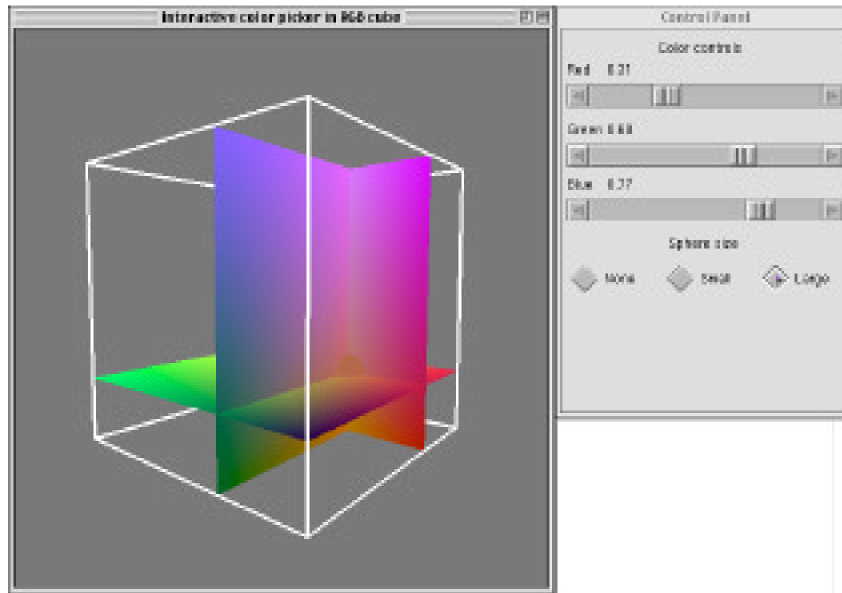

Figure: the color selector in context, with both the display and control windows shown

A word to the wise...

The MUI control window has behaviors that are outside the programmer's control, so you must be aware of some of these in order to avoid some surprises. The primary behavior to watch for is that many of the MUI elements include a stream of events (and their associated redisplays) whenever the cursor is within the element's region of the window. If your application was not careful to insulate itself against changes caused by redisplays, you may suddenly find the application window showing changes when you are not aware of requesting them or of creating any events at all. So if you use MUI, you should be particularly conscious of the structure of your application on redisplay and ensure that (for example) you clear any global variable that causes changes in your display before you leave the display function.

Reference

Steve Baker, A Brief MUI User Guide, distributed with the MUI release

Code examples

- `glutColors.c` — source for the example in this section