

III.2. The CADshell Windows Libraries

Most of the common window types have been encapsulated into classes in the “CADshell” Windows library. This library is part of a larger CAD research project. These classes are similar, but simpler than those found in other Windows toolkits such as Visual C++, or X-Windows. Classes belonging to the CADshell Windows library have their names prefixed with “sh_”. These classes are described with examples in the files controls.h and controls_main.h. Here is an example program that simply pops up a main window with some text.

```
int main()
{
    sh_MainWindow *main_window = new sh_MainWindow("My
        Application");
    sh_Label *label = new sh_Label(main_window, "Hello world! This
        is xxx.");
    main_window -> Run();
    return 0;                // never reached
}
```

To add a button, use:

```
int main()
{
    sh_MainWindow *main_window = new sh_MainWindow("My
        Application");
    sh_Label *label = new sh_Label(main_window, "Hello world! This
        is xxx.");
    sh_Button *button = new sh_Button(main_window, "Press me!");
    button->IsBelow(label);
    main_window -> Run();
    return 0;                // never reached
}
```

Look at the sh_Control class declaration in controls.h to see other positioning commands that are available.

To add a menu, it is necessary to:

- (i) create an sh_MenuBar object with main_window as parent,
- (ii) create sh_PulldownMenu objects with the menu bar as parent,
- (iii) create sh_Button objects with sh_PulldownMenu as parent.

```

int main()
{
    sh_MainWindow *main_window = new sh_MainWindow("My
        Application");
    sh_MenuBar *menubar = new sh_MenuBar(main_window);
    sh_PulldownMenu *filePulldown = new sh_PulldownMenu(menubar,
        "File", 'F');
    sh_PulldownMenu *editPulldown = new sh_PulldownMenu(menubar,
        "Edit", 'E');
    sh_Button *fileopen = new sh_Button(filePulldown, "Open");
    sh_Label *label = new sh_Label(main_window, "Hello world! This
        is xxx.");
    label->IsBelow(menubar);
    sh_Button *button = new sh_Button(main_window, "Press me!");
    button->IsBelow(label);
    main_window -> Run();
}

```

The constructor parameters are described in controls_main.h.

This program now has a menu and two buttons, but there is no functionality attached to these controls. To add this functionality we need to create event handlers to handle events. The types of events that can be handled depend on the type of control:

ValueChanged – the user has manipulated the control (all sh_Control’s)

ControlClosed – the user has closed the window (all sh_Control’s)

Apply – a new value has been accepted by the user (all sh_Control’s)

ButtonPressed – button was pressed (sh_Button controls)

SliderSlid – the slider control was moved by the user (sh_Slider)

ScrollbarSlid – scrollbar was moved by the user (sh_Scrollbar)

In this course, objects of classes derived from sh_EventHandler perform the event handling.

Let’s create a handler “ButtonEvent” to print a message when the button is pressed.

```

class ButtonEvent : public sh_Tool
{
public:
    void ControlEvent(sh_Control *, sh_Event &);
};

```

```

void ButtonEvent::ControlEvent(sh_Control *, sh_Event &)
{
    cout << "The button was pressed!" << endl;
}

```

The main() function must also be modified to add in the ButtonEvent event handler to the button control.

```

...
sh_Button *button = new sh_Button(main_window, "Press me!", new
    ButtonEvent);
...

```

The sh_EventHandler function ControlEvent() is called whenever the user manipulates the control. In this case, the sh_Button::ControlEvent() function will get called by the button control object every time the button is pressed by the user. The function has been re-defined to suit our purposes. The first parameter in this function tells the event handler in which control the event occurred. The second parameter tells the event handler more information about the event. We do not need the information from these parameters in our function.

Note that in this example, it is not necessary to delete the ButtonEvent object since it is active until the program terminates.

In the previous example, we printed a message to the console. Normally, a Windows program does not print to the console. Instead, message boxes are used. The ButtonEvent::ControlEvent() function can be re-written as:

```

void ButtonEvent::ControlEvent(sh_Control *, sh_Event &)
{
    sh_MessageBox::Message("The button was pressed!");
}

```

This event handler causes a message box to pop up with the message "The button was pressed". Message() is a static function of the sh_MessageBox class, so it can be called without creating an object.

Instead of creating a message box, we can create a dialog box to input the value of an integer.

```

void ButtonEvent::ControlEvent(sh_Control *, sh_Event &)
{
    int integer_to_input = 0;
    sh_Dialog *d = new sh_Dialog("Input Integer");
}

```

```

sh_Control *c1 = new sh_IntInput(d, &integer_to_input);
sh_Control *c2 = new sh_OkApplyCancel(d);
c2->IsBelow(c1);

if (d->Execute() == IDOK)
{
    stringstream msg;
    msg << "The number is: " << integer_to_input;
    char *str = msg.str();
    sh_MessageBox::Message(str);
    free (str);
}
}

```

The first line of the function defines the integer variable and initializes it to 0. The second line instantiates a dialog box object. This is almost like instantiating a main window object. The next two lines create two different controls: a window to input the integer value and a window that contains Ok, Apply and Cancel buttons. Note that for each window we give the dialog box as the parent in the first parameter of the constructors and we also pass in a pointer to the integer variable in the `sh_IntInput` constructor. The control `c2` is positioned below `c1`. The `sh_Dialog::Execute()` function is then called for `d`, which causes the dialog box to pop up. This function does not return until the user presses “Ok” or “Cancel”. If the user has pressed “Ok”, then the `Execute()` function returns `IDOK`. Other return values are described in the file `controls.h`. If `Execute()` returns `IDOK`, the message is printed to the `msg` variable and displayed in the message box. The `msg` variable is an object of the `stringstream` class and acts just like `cout`, however, the message is printed to a string, which is accessed using the `stringstream::str()` function. To use the `stringstream` class, you must include the `stringstream.h` header file.

A dialog can be “modal” or “modeless”. A “modal” dialog will not allow the user to perform any other actions until the dialog box has been responded to and is closed. A “modeless” dialog can be set aside for later consideration.

To create a modal dialog, use `Execute()`. To create a modeless dialog, use `Create()` instead. `Create()` is different from `Execute()` in that it returns as soon as the dialog box has popped up. An event handler must therefore be setup to handle the user responses from the dialog box. This complicates the code significantly.

Using pre-defined dialog boxes is similar to using dialog boxes you build up yourself. To use the `sh_FileDialog` box, first create an event handler “FileOpen” to handle the File Open menu selection:

```

class FileOpen : public sh_EventHandler
{
public:
    void ControlEvent(sh_Control *, sh_Event &);
};

```

```

void FileOpen::ControlEvent(sh_Control *, sh_Event &)
{
    char *filename = 0;
    sh_FileDialog *dlg = new sh_FileDialog("File Open", &filename);
    if (dlg->Execute() == IDOK)
        cout << "We're going to open file: " << filename << endl;
    if (filename)
        free (filename);
}

```

The main() function must also be modified so that the FileOpen event handler is added to the fileopen button.

```

...
sh_Button *fileopen = new sh_Button(filePulldown, "Open", new
    FileOpen);
...

```

To make the FileOpen::ControlEvent() function actually read in a file, we would need to add other code according to the format of the file and how the program stores the data.

To further understand the Event Handling process, here is a program that calls the sh_Control::TransferFromControl() function to read in a double value after a button is pressed:

```

class ButtonEventHandler : public sh_EventHandler
{
public:
    ButtonEventHandler(double *anum) { num_pointer = anum; }
    void ControlEvent(sh_Control *, sh_Event &);
protected:
    double *num_pointer;
};

void ButtonEventHandler::ControlEvent(sh_Control *, sh_Event &)
{
    sh_MainWindow::GetMainWindow()->TransferFromControl();
    cout << "The number is now :" << *num_pointer << endl;
}

int main()
{
    double num = 0.;
    sh_MainWindow *main_win = new sh_MainWindow("Calculator");
    sh_DoubleInput *num_input = new sh_DoubleInput(main_win, &num);
}

```

```
sh_Button *button = new sh_Button(main_win, "press to  
transfer", new ButtonEventHandler(&num));  
button->IsBelow(num_input);  
main_win->Run();  
return 0;  
}
```