Advanced Windows Programming

Aug 12, 2002

This chapter is from the book

Tricks of the Windows Game Programming Gurus, 2nd Edition

Learn More Buy

Working with Menus

Menus are one of the coolest things about a Windows program and are ultimately the point of interaction between the user and your program (that is, if you're making a word processor <BG>). Knowing how to create and work with menus is very important because you might want to design simple tools to help create your game, or you might want to have a window-based front end to start up your game. And these tools will undoubtedly have menus—millions of them if you're making a 3D tool. Trust me! In either case, you need to know how to create, load, and respond to menus.

Creating a Menu

You can create an entire menu and all the associated files with the compiler's menu editor, but we'll do it manually because I can't be sure which compiler you're using. This way you'll learn what's in a menu description, too. But when you're writing a real application and creating a menu, most of the time you'll use the IDE editor because menus are just too complex to type in manually. It's like HTML code—when the Web started, it wasn't a big deal to make a home page with a text editor. Nowadays, it's nearly impossible to create a Web site without using a tool. (Speaking of Web site design, my friend needs work at http://www.belmdesigngroup.com—he has 15 kids to feed!)

Anyway, let's get started making menus! Menus are just like the other resources you have already worked with. They reside in an .RC resource script and must have an .H file to resolve any symbolic references, which are all IDs in the case of menus. (One exception: The name of the menu must be symbolic—no name strings.) Here's the basic syntax of a MENU description as you would see it in an .RC file:

```
MENU_NAME MENU DISCARDABLE
{ // you can use BEGIN instead of { if you wish

// menu definitions

} // you can use END instead of } if you wish
```

MENU_NAME can be a name string or a symbol, and the keyword DISCARDABLE is vestigial
necessary. Seems simple enough. Of course, the stuff in the middle is missing, but chill—I'r

there!

Before I show you the code to define menu items and submenus, we need to get some terminology straight. For my little discussion, refer to the menu in Figure 3.9. It has two top-level menus, File and Help. The File menu contains four menu items: Open, Close, Save, and Exit. The Help menu contains only one menu item: About. So there are top-level menus and menu items within them. However, this is misleading because it's possible to also have menus within menus, or *cascading menus.* I'm not going to create any cascading menus, but the theory is simple: You just use a menu definition for one of the menu items itself. You can do this recursively, ad infinitum.

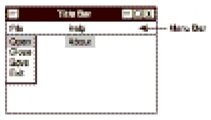

Figure 3.9 A menu bar with two submenus.

Now that we have the terminology straight, here's how you would implement the menu shown in Figure 3.9:

```
MainMenu MENU DISCARDABLE
{
POPUP "File"
  {
  MENUITEM "Open", MENU_FILE_ID_OPEN
  MENUITEM "Close", MENU_FILE_ID_CLOSE
  MENUITEM "Save", MENU_FILE_ID_SAVE
  MENUITEM "Exit", MENU_FILE_ID_EXIT
  } // end popup

POPUP "Help"
  {
  MENUITEM "About", MENU_HELP_ABOUT
  } // end popup

} // end top level menu
```

Let's analyze the menu definition section by section. To begin with, the menu is named MainMenu. At this point we don't know if it's a name string or an ID, but since I usually capitalize all constants, it's a safe bet that it's a plain string. So that's what we'll make it. Moving on, there are two top-level menu definitions, beginning with the keyword POPUP—this is key. POPUP indicates that a menu is being defined with the following ASCII name and menu items.

The ASCII name must follow the keyword POPUP and be surrounded by quotes. The pop-up menu definition must be contained within `{ }` or a `BEGIN END` block—whichever you like. (You Pascal people should be happy <BG>.)

Within the definition block, follow all of the menu items. To define a menu item, you use the keyword `MENUITEM` with the following syntax:

```
MENUITEM "name", MENU_ID
```

And that's it! Of course, in this example you haven't defined all the symbols, but you would do so in an .H file something like this:

```
// defines for the top level menu FILE
#define MENU_FILE_ID_OPEN        1000
#define MENU_FILE_ID_CLOSE       1001
#define MENU_FILE_ID_SAVE        1002
#define MENU_FILE_ID_EXIT        1003

// defines for the top level menu HELP
#define MENU_HELP_ABOUT         2000
```

TIP

Notice the values of the IDs. I have selected to start off the first top-level menu at 1000 a
increment by 1 for each item. Then I increment by 1000 for the next top-level menu. Thu
top-level menu differs by 1000, and each menu item within a menu differs by 1. This is a
convention that works well. And it's less filling.

I didn't define `"MainMenu"` because I want to refer to the menu by string rather than ID. This isn't the only way to do it. For example, if I put the single line of code

```
#define MainMenu 100
```

within the .H file with the other symbols, the resource compiler would automatically assume that I wanted to refer to the menu by ID. I would have to use MAKEINTRESOURCE(MainMenu) or MAKEINTRESOURCE(100) to refer to the menu resource. Get it? Alrighty, then!

NOTE

You'll notice that many menu items have hotkeys or shortcuts that you can use instead of manually selecting the top-level menu or menu item with the mouse. This is achieved by using the ampersand character (&). All you do is place the ampersand in front of the character that you want to be a shortcut or hotkey in a POPUP menu or a `MENUITEM` string. For example,

```
MENUITEM "E&xit", MENU_FILE_ID_EXIT
```

makes the x a hotkey, and

```
POPUP "&File"
```

makes F a shortcut via Alt+F.

Now that you know how to create and define a menu, let's see how to load it into your application and attach it to a window.

Loading a Menu

There are a number of ways to attach a menu to a window. You can associate a single menu with all windows in a Windows class, or you can attach different menus to each window that you create. First, let's see how to associate a single menu with the Windows class itself.

In the definition of the Windows class, there is a line of code that defines what the menu is

```
winclass.lpszMenuName  = NULL;
```

All you need to do is assign it the name of the menu resource. Presto, that's it! Here's how

```
winclass.lpszMenuName  = "MainMenu";
```

And if "MainMenu" was a constant, you would do it this way:

```
winclass.lpszMenuName  = MAKEINTRESOURCE(MainMenu);
```

No problemo...almost. The only problem with this is that every window you create will have the same menu. To get around this, you can assign a menu to a window during creation by passing a menu handle. However, to get a menu handle, you must load the menu resource with LoadMenu(). Here's its prototype(s):

```
HMENU LoadMenu(HINSTANCE hInstance,// handle of application instanc
 LPCTSTR lpMenuName);// menu name string or menu-resource identifie
```

If successful, LoadMenu() returns an HMENU handle to the menu resource, which you can

Here's the normal `CreateWindow()` call you have been making, changed to load the mer
`"MainMenu"` into the menu handle parameter:

```
// create the window
if (!(hwnd = CreateWindowEx(NULL,         // extended style
          WINDOW_CLASS_NAME,  // class
         "Sound Resource Demo", // title
          WS_OVERLAPPEDWINDOW | WS_VISIBLE,
          0,0,   // initial x,y
          400,400, // initial width, height
          NULL,   // handle to parent
          LoadMenu(hinstance, "MainMenu"), // handle to menu
          hinstance,// instance of this application
          NULL)))  // extra creation parms
return(0);
```

Or if MainMenu was a symbolic constant, the call would look like this:

```
LoadMenu(instance, MAKEINTRESOURCE(MainMenu)), // handle to menu
```

NOTE

You may think I'm belaboring the difference between resources defined by string and by symbolic constant. However, taking into consideration that it's the number one cause of self-mutilation among Windows programmers, I think it's worth the extra work—don't you?

And of course, you can have many different menus defined in your `.RC` file, and thus you can attach a different one to each window.

The final method of attaching a menu to a window is by using the `SetMenu()` function, shown here:

```
BOOL SetMenu(HWND hWnd,  // handle of window to attach to
       HMENU hMenu); // handle of menu
```

SetMenu() takes the window handle, along with the handle to the menu (retrieved from LoadMenu()), and simply attaches the menu to the window. The new menu will override any menu previously attached. Here's an example listing, assuming that the Windows class defines the menu as NULL, as does the menu handle in the call to CreateWindow():

```
// first fill in the window class structure
winclass.cbSize = sizeof(WNDCLASSEX);
winclass.style   = CS_DBLCLKS | CS_OWNDC |
               CS_HREDRAW | CS_VREDRAW;
winclass.lpfnWndProc  = WindowProc;
winclass.cbClsExtra   = 0;
winclass.cbWndExtra  = 0;
winclass.hInstance    = hinstance;
winclass.hIcon       = LoadIcon(hinstance,
                   MAKEINTRESOURCE(ICON_T3DX));
winclass.hCursor     = LoadCursor(hinstance,
                   MAKEINTRESOURCE(CURSOR_CROSSHAIR));
winclass.hbrBackground  = (HBRUSH)GetStockObject(BLACK_BRUSH);
winclass.lpszMenuName  = NULL; // note this is null
winclass.lpszClassName  = WINDOW_CLASS_NAME;
winclass.hIconSm   = LoadIcon(hinstance, MAKEINTRESOURCE(ICON_T3DX));

// register the window class
if (!RegisterClassEx(&winclass))
  return(0);

// create the window
if (!(hwnd = CreateWindowEx(NULL,       // extended style
        WINDOW_CLASS_NAME, // class
        "Menu Resource Demo", // title
        WS_OVERLAPPEDWINDOW | WS_VISIBLE,
        0,0,   // initial x,y
        400,400, // initial width, height
        NULL,   // handle to parent
        NULL,   // handle to menu, note it's null
        hinstance,// instance of this application
        NULL)))  // extra creation parms
return(0);

// since the window has been created you can
// attach a new menu at any time
// load the menu resource
HMENU hmenuhandle = LoadMenu(hinstance, "MainMenu");

// attach the menu to the window
SetMenu(hwnd, hmenuhandle);
```

For an example of creating the menu and attaching it to the window using the second method (that is, during the window creation call), take a look at DEMO3_3.CPP on the CD-ROM and the associated executable, DEMO3_3.EXE, which is shown running in Figure 3.10.

Figure 3.10 Running DEMO3_3.EXE.

The only two files of interest are the resource and header files, DEMO3_3RES.H and DEMO3_3.RC.

Contents of DEMO3_3RES.H:

```
// defines for the top level menu FILE
#define MENU_FILE_ID_OPEN        1000
#define MENU_FILE_ID_CLOSE       1001
#define MENU_FILE_ID_SAVE        1002
#define MENU_FILE_ID_EXIT        1003

// defines for the top level menu HELP
#define MENU_HELP_ABOUT         2000
```

Contents of DEMO3_3.RC:

```
#include "DEMO3_3RES.H"

MainMenu MENU DISCARDABLE
{
POPUP "File"
  {
  MENUITEM "Open", MENU_FILE_ID_OPEN
  MENUITEM "Close", MENU_FILE_ID_CLOSE
  MENUITEM "Save", MENU_FILE_ID_SAVE
  MENUITEM "Exit", MENU_FILE_ID_EXIT
  } // end popup

POPUP "Help"
  {
  MENUITEM "About", MENU_HELP_ABOUT
  } // end popup

} // end top level menu
```

To compile your own DEMO3_3.CPP executable, make sure to include

DEMO3_3.CPP—The main source.
DEMO3_3RES.H—The resource symbol header.
DEMO3_3.RC—The resource script file.
Try playing with DEMO3_3.EXE and the associated source. Change the menu items, add some more menus by adding more POPUP blocks to the .RC file, and get a good feel for it. Also, try making a cascading menu tree. (Hint: Just replace MENUITEM with a POPUP for one of the MENUITEMS making up a menu.)

Responding to Menu Event Messages

The only problem with DEMO3_3.EXE is that it doesn't do anything! True, my young Jedi. The problem is that you don't know how to detect the messages that menu item selections and manipulations generate. That is the topic of this section.

The Windows menu system generates a number of messages as you slide across top-level items as shown in Figure 3.11.

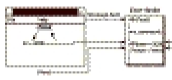
Figure 3.11 Window menu selection message flow.

The message we are interested in is sent when a menu item is selected and then the mous released. This denotes a *selection.* Selections send a WM_COMMAND message to the WinPrc the window that the menu is attached to. The particular menu item ID and various other data are stored in the wparam and lparam of the message, as shown here:

msg—WM_COMMAND
lparam—The window handle that the message was sent from
wparam—The ID of the menu item that was selected

TIP

Technically, you should extract the low-order WORD from wparam with the LOWORD() macro to be safe. This macro is part of the standard includes, so you have access to it.

So all you have to do is switch() on the wparam parameter, with the cases being the different MENUITEM IDs defined in your menu, and you're in business. For example, using the menu defined in DEMO3_3.RC, you would add the WM_COMMAND message handler and end up with something like this for your WinProc():

```
LRESULT CALLBACK WindowProc(HWND hwnd,
              UINT msg,
              WPARAM wparam,
              LPARAM lparam)
{
// this is the main message handler of the system
PAINTSTRUCT    ps;  // used in WM_PAINT
HDC            hdc;  // handle to a device context

// what is the message
switch(msg)
  {
  case WM_CREATE:
    {
  // do initialization stuff here

    // return success
    return(0);
  } break;

    case WM_COMMAND:
    {
    switch(LOWORD(wparam))
       {
       // handle the FILE menu
       case MENU_FILE_ID_OPEN:
       {
       // do work here
       } break;
       case MENU_FILE_ID_CLOSE:
       {
       // do work here
       } break;
       case MENU_FILE_ID_SAVE:
       {
       // do work here
       } break;
       case MENU_FILE_ID_EXIT:
       {
       // do work here
       } break;

       // handle the HELP menu
       case MENU_HELP_ABOUT:
       {
       // do work here
       } break;

       default: break;

       } // end switch wparam

    } break; // end WM_COMMAND

  case WM_PAINT:
  {
  // simply validate the window
  hdc = BeginPaint(hwnd,&ps);
  // you would do all your painting here
    EndPaint(hwnd,&ps);
    // return success
  return(0);
  } break;

  case WM_DESTROY:
  {
  // kill the application, this sends a WM_QUIT message
  PostQuitMessage(0);

    // return success
  return(0);
  } break;

    default:break;

  } // end switch

// process any messages that we didn't take care of
return (DefWindowProc(hwnd, msg, wparam, lparam));

} // end WinProc
```

It's so easy, it should be illegal! Of course, there are other messages that manipulate the top-level menus and menu items themselves, but you can look in your Win32 SDK Help for more info. (I rarely need to know more than if a menu item was clicked or not.)

As a solid example of doing something with menus, I have created a cool sound demo that allows you to exit the program via the main menu, play one of four different teleporter sound effects, and finally pop up an About box via the Help menu. Also, the .RC file contains the sound, icon, and cursor resources. The program is DEMO3_4.CPP. Let's take a look at the resource script and header first.

Contents of DEMO3_4RES.H:

```
// defines for sounds resources
#define SOUND_ID_ENERGIZE  1
#define SOUND_ID_BEAM    2
#define SOUND_ID_TELEPORT  3
#define SOUND_ID_WARP  4

// defines for icon and cursor
#define ICON_T3DX    100
#define CURSOR_CROSSHAIR 200

// defines for the top level menu FILE
#define MENU_FILE_ID_EXIT        1000

// defines for play sound top level menu
#define MENU_PLAY_ID_ENERGIZE      2000
#define MENU_PLAY_ID_BEAM        2001
#define MENU_PLAY_ID_TELEPORT      2002
#define MENU_PLAY_ID_WARP        2003

// defines for the top level menu HELP
#define MENU_HELP_ABOUT         3000
```

Contents of DEMO3_4.RC:

```
#include "DEMO3_4RES.H"

// the icon and cursor resource
ICON_T3DX    ICON  t3dx.ico
CURSOR_CROSSHAIR CURSOR crosshair.cur

// the sound resources
SOUND_ID_ENERGIZE  WAVE energize.wav
SOUND_ID_BEAM    WAVE beam.wav
SOUND_ID_TELEPORT  WAVE teleport.wav
SOUND_ID_WARP    WAVE warp.wav

// the menu resource
SoundMenu MENU DISCARDABLE
{
POPUP "&File"
  {
  MENUITEM "E&xit", MENU_FILE_ID_EXIT
  } // end popup

POPUP "&PlaySound"
  {
    MENUITEM "Energize!",      MENU_PLAY_ID_ENERGIZE
    MENUITEM "Beam Me Up",      MENU_PLAY_ID_BEAM
    MENUITEM "Engage Teleporter",  MENU_PLAY_ID_TELEPORT
    MENUITEM "Quantum Warp Teleport", MENU_PLAY_ID_WARP
  } // end popup
POPUP "Help"
  {
  MENUITEM "About", MENU_HELP_ABOUT
  } // end popup

} // end top level menu
```

Based on the resource script and header file (which must be included in the main app), let's take a look at the code excerpts of DEMO3_4.CPP loading each resource. First, the loading of the main menu, icon, and cursor:

```
winclass.hCursor = LoadCursor(hinstance,
           MAKEINTRESOURCE(CURSOR_CROSSHAIR));
winclass.lpszMenuName = "SoundMenu";
winclass.hIcon  = LoadIcon(hinstance, MAKEINTRESOURCE(ICON_T3DX));
winclass.hIconSm= LoadIcon(hinstance, MAKEINTRESOURCE(ICON_T3DX));
```

And now the fun part—the processing of the WM_COMMAND message that plays each sound, along with the handling of the Exit menu item and the display of the About box under Help. For brevity, I'll just show the WM_COMMAND message handler, since you've seen the entire WinProc() enough by now:

```
case WM_COMMAND:
    {
    switch(LOWORD(wparam))
       {
       // handle the FILE menu
       case MENU_FILE_ID_EXIT:
       {
       // terminate window
       PostQuitMessage(0);
       } break;

       // handle the HELP menu
       case MENU_HELP_ABOUT:
       {
       // pop up a message box
       MessageBox(hwnd, "Menu Sound Demo",
               "About Sound Menu",
               MB_OK | MB_ICONEXCLAMATION);
       } break;
       // handle each of sounds
       case MENU_PLAY_ID_ENERGIZE:
       {
       // play the sound
       PlaySound(MAKEINTRESOURCE(SOUND_ID_ENERGIZE),
            hinstance_app, SND_RESOURCE | SND_ASYNC);
       } break;
       case MENU_PLAY_ID_BEAM:
       {
       // play the sound
       PlaySound(MAKEINTRESOURCE(SOUND_ID_BEAM),
            hinstance_app, SND_RESOURCE | SND_ASYNC);
       } break;
       case MENU_PLAY_ID_TELEPORT:
       {
       // play the sound
       PlaySound(MAKEINTRESOURCE(SOUND_ID_TELEPORT),
            hinstance_app, SND_RESOURCE | SND_ASYNC);
       } break;
       case MENU_PLAY_ID_WARP:
       {
       // play the sound
       PlaySound(MAKEINTRESOURCE(SOUND_ID_WARP),
            hinstance_app, SND_RESOURCE | SND_ASYNC);
       } break;

       default: break;
       } // end switch wparam
   } break; // end WM_COMMAND
```

And that's all I have to say about that.

As you can see, resources can do a lot and are fun to work with. Now let's take a break fror resources and take an introductory crash course on the WM_PAINT message and basic GDI manipulation.

Save To Your Account

< Back Page 2 of 6 Next >