CHAPTER 9

UI Programming

Previous chapters have focused on modifying the `game` code and changing game features such as how weapons behave, how players move, and how visual effects are created. In this chapter, you'll make a departure from the `game` logic, and take a look at another important unit of the *Q3* source: the `ui` module, which controls the user interface. The most exciting and innovative game in the world is nothing if the user cannot configure it to his liking. In this chapter, I plan to show you the various elements that comprise the `ui` code, and demonstrate how they interact with one another, building the menu systems that you use when you set up your player's preferences, controls, display settings, and so on.

Basic UI Concepts

In order to modify the `ui` code, a few introductions are in order. First, you must understand the basic system upon which the user interface is built. Many of the objects you will be looking at in the `ui` code resemble similar sorts of implementations across many Win32-based applications. The user interface typically consists of *menus*—virtual pages of controls that have various formats applied to them, which, in turn, affect how those particular controls are displayed onscreen. As well, controls can have *events* assigned to them, causing certain functions to execute when a control is made active or inactive, or is being changed by the user. Once you are familiar with the specific terminology behind these descriptions, you'll have an easier time visualizing your own user-interface designs.

NOTE

Although I'll be referring to the user-interface code as `ui` throughout this chapter, I'll be dealing specifically with the `q3_ui` project, so be sure that the files you modify are in this project. Otherwise, you will be modifying the user-interface code for the Team Arena Expansion Pack!

Controls: Nuts and Bolts of UI

As in many Win32 applications, a user interface is simply a collection of controls that are organized across one or more pages or *layouts*. These layouts allow the user to manipulate settings on the system on which the UI is based. If you've ever filled out a form on Web, or played around with your computer system's settings, you will have undoubtedly used multiple controls in the process. Boxes in which you can type text, sliders, drop-down menus, and buttons are all examples of controls.

As expected, *Q3* has its own set of controls that are implemented across a various set of menus, which allow players to configure the game to their particular tastes. There are seven controls in total. Table 9.1 lists these controls by their native C-style struct declarations, followed by descriptions of the controls.

Table 9.1 *Q3* UI Controls

Name	Description
menufield_s	This control allows text to be entered via a rectangular box.
menuslider_s	This control features a bar that represents a range of values. A slider arrow or *thumb* can then be used to select various values along the range.
menulist_s	This control shows a list of items to be scrolled through, allowing a specific item to be selected.
menuradiobutton_s	This control is used to specify that specific data be either active or inactive.
menubitmap_s	This control is used to represent buttons or images.
menutext_s	This control allows read-only text to be displayed on-screen; it does not allow the user to change the text.
menuaction_s	This control allows for a larger amount of text to be displayed on-screen, also in a read-only format.

The developers at id designed the user-interface code to mirror many familiar techniques already used in Win32-based development. To see some of these controls in action, take a peek at Figure 9.1, which shows one menu in *Q3*'s user interface.

In this image, there is an instance of the menufield_s control, which allows the user to type the name of his player. There is also an example of the menulist_s control, which allows the user to select one of many handicaps (or none at all). There are many layouts of menus in the `ui` code that follow this same simple rule: Allow the user to make changes to the game through the use of controls.

Formatting Controls

Although it may sound fine and dandy to have a control that allows the user to type some arbitrary text into it, you have to admit that text controls aren't terribly exciting. Your apathy will likely be compounded when you take into consideration the fact that, functionally,

Figure 9.1 *The player settings menu in* Q3

there isn't a lot of room for customization of controls. As is traditional in most UI-programming APIs, however, controls can be altered or *formatted* in certain ways to meet the demands of the programmer; *Q3*'s UI is no exception.

Each control available to you in the `ui` code can have a certain formatting style associated with it. This is done through the use of bit flags, with which you have some experience by now. Typically, formatting bit flags can be applied to controls in two ways: during initialization, meaning that the format is applied to the control for its entire duration, or during existence, meaning that the format can change the style of the control on-the-fly while the user interacts with it. Take a look at Table 9.2, which lists the existing menu-formatting flags.

NOTE

***API* stands for *Application Programming Interface*. An API represents a common set of functions that a specific application can use to complete lower-level tasks, often easing the programmer's workload. Because the *Q3* `ui` code encapsulates or *hides* a lot of the dirty work necessary to set up a menu system, it fits the definition of an API.**

NOTE

Some of these formatting flags do not apply to all controls, such as `QMF_LEFT_JUSTIFY`, `QMF_CENTER_JUSTIFY`, and `QMF_RIGHT_JUSTIFY`. Also, some bit flags cannot be mixed and matched with each other. `QMF_HIGHLIGHT`, for example, cannot be combined with `QMF_BLINK`; a control is either highlighted or is blinking, never both at once.

Now you should be able to see how controls can be tweaked and modified so that they are more flexible for the developer.

Controls Have One Thing in menucommon_s

Each of the controls in the `ui` code is built upon a generic set of data (which, amusingly enough, is defined with the variable name `generic` in each control). The data that is common to each control is held in a

Table 9.2 Menu-Formatting Flags

Name	Description
QMF_BLINK	This flag causes text to flash on and off.
QMF_SMALLFONT	This flag causes text to be drawn in a small font.
QMF_LEFT_JUSTIFY	This flag positions the control flush to the left.
QMF_CENTER_JUSTIFY	This flag centers the control.
QMF_RIGHT_JUSTIFY	This flag positions the control flush to the right.
QMF_NUMBERSONLY	This flag restricts data entry to numerical values only.
QMF_HIGHLIGHT	This flag renders the control brighter, giving it more presence on the menu.
QMF_HIGHLIGHT_IF_FOCUS	This flag renders the control brighter if it is the control being activated by the user.
QMF_PULSEIFFOCUS	This flag causes the control to fade in and out if it is the control being activated by the user.
QMF_HASMOUSEFOCUS	This flag is read-only, and exists on any control that currently has the mouse pointer hovering over it.
QMF_MOUSEONLY	This flag disallows the control from being activated by the keyboard.
QMF_HIDDEN	This flag hides the control from view.
QMF_GRAYED	This flag renders the control in a darker color, signifying that it is an unusable control.
QMF_INACTIVE	This flag disallows user input. It is applied by default to controls carrying the QMF_GRAYED flag.
QMF_NODDEFAULTINIT	This flag prevents *Q3* from automatically handling initialization of the control. It is used by controls not already defined in the *Q3* UI.
QMF_PULSE	This flag causes the control to fade in and out.
QMF_LOWERCASE	This flag causes text entered into the control to be all lowercase.
QMF_UPPERCASE	This flag causes text entered into the control to be all uppercase.
QMF_SILENT	This flag indicates to *Q3* that no sound is to be played when the control is activated.

struct called *menucommon_s*, which is declared on line 143 on ui_local.h. Here is that structure of data:

```
typedef struct
{
    int     type;
    const  char *name;
    int     id;
    int     x, y;
    int     left;
    int     top;
    int     right;
    int     bottom;
    menuframework_s *parent;
    int menuPosition;
    unsigned flags;

    void (*callback)( void *self, int event );
    void (*statusbar)( void *self );
    void (*ownerdraw)( void *self );
} menucommon_s;
```

As is indicative of this struct, a control in the `ui` will be of a certain type, and will have a name to describe the control. An ID is also used to help identify the control and keep it unique in a menu (because there is no constraint on the number of similar controls per menu). The `x` and `y` members relate to where the control resides on the screen. The next four members, `left`, `top`, `right`, and `bottom`, represent the bounding box for the control, which can be used to detect whether the user's mouse has entered a specific control's area. You can see the `flags` member, which represents the formatting flags that can be applied to the control.

The final three members are pointers to functions. The first, `callback`, represents the activity that the control will carry out when it is used. `statusbar`, the

NOTE

I skipped over the `*parent` pointer because I have not yet dealt with the menuframework_s struct, but I will be covering it shortly. As for `menuPosition`, it is a variable that does not need to be set or updated by you in any capacity, so it is safe to ignore.

next member, is used to display additional data if the control detects a mouse pointer in its bounding box. The final member, ownerdraw, is used to extend the flexibility of the control, via a custom function.

The common data held in menucommon_s serves as a basis for each control, which can then be built upon with specific unique data for each individual control. Later in this chapter you will look at the specific data for each control.

The Menu Framework

Now that you have a handle on the structure of a control, you need to know how to place that control into a menu. In the ui code, controls are added to menus through the use of a struct called *menuframework_s*. The menuframework_s struct sits alongside the controls that are needed on a given menu; by combining menuframework_s with a specific set of controls, you can instantiate your own user interface. The core of menuframework_s is declared on line 127 of ui_local.h.

```
typedef struct _tag_menuframework
{
    int     cursor;
    int     cursor_prev;

    int     nitems;
    void    *items[MAX_MENUITEMS];

    void    (*draw) (void);
    sfxHandle_t (*key) (int key);

    qboolean    wrapAround;
    qboolean    fullscreen;
    qboolean    showlogo;
} menuframework_s;
```

You will want to concern yourself with the initialization of three members in this struct. The first variable is wrapAround, which is a qboolean that determines whether the menu allows the user to scroll through its list of controls indefinitely. For example, suppose a user was cycling through a list of menu choices with his arrow keys and he reached the

last menu item. If `wrapAround` were set to `qtrue`, then first item on the menu would be selected the next time the user pressed his down-arrow key. A value of `qfalse`, on the other hand, represents a hard beginning and ending to a list of controls.

The second member to be initialized is `fullscreen`, which is a qboolean that specifies how the menu handles the activity in the game when accessed. If `fullscreen` is set to `qtrue`, the menu will pause the game currently being played. A setting of `qfalse` will allow the game to continue in the background while the menu is being accessed.

The final member of importance is the function pointer `draw`. This function is used to allow more controls to be added to the current menu for rendering. It is worth mentioning that additional controls drawn by this function would have to be cached ahead of time. Let's take a look at a current menu in *Q3* using the menuframework_s struct. I'll pick an easy menu for you to visualize: the main menu that is presented when you first load *Q3*.

```
typedef struct {
        menuframework_s      menu;

        menutext_s           singleplayer;
        menutext_s           multiplayer;
        menutext_s           setup;
        menutext_s           demos;
        menutext_s           cinematics;
        menutext_s           teamArena;
        menutext_s           mods;
        menutext_s           exit;

        qhandle_t            bannerModel;
} mainmenu_t;
```

This (as found on line 28 of ui_menu.c) is the menu construct for the very first interface. As you can see, menuframework_s is the very first member of the mainmenu_t struct. Every menu is designed in this manner—you'll want to remember that when it comes time to make your own menu. Following the menuframework_s member is a series of controls; in this case, they all happen to be menutext_s—except for the last member, which is a qhandle_t.

Let's take a look at this menutext_s control and see what makes it tick. The declaration of a menutext_s struct is on line 227 of ui_local.h.

```
typedef struct
{
    menucommon_s    generic;
    char*           string;
    int             style;
    float*          color;
} menutext_s;
```

TIP

If you have a sharp memory, you'll recall that a qhandle_t was covered in the Chapter 6. The qhandle_t in this struct references the shimmering 3D logo that hovers along the top of the menu, spelling out the words "Quake III Arena."

Not too complicated, by the looks of things. A menutext_s control, as mentioned earlier, is used to display some static or *unmodifiable* text on the screen. The menutext_s control is actually quite flexible despite its simplicity, and comes in three popular flavors. The first of these styles is a straight-up, no-nonsense string of text, without any crazy options or special effects. The characters used to render the text are a fixed width, and the font-size bit flags applied to the `style` member determines their size. For this vanilla text control, the `generic.type` variable is set to `MTYPE_TEXT`.

You can also use the menutext_s control to render text to the screen in a banner style, which simply draws the text larger and with a proportional font. This is the perfect type of control to use as the name of a menu's section, or the header of an important part of your menu. To achieve this effect, you assign `generic.type` a value of `MTYPE_BTEXT`.

The final flavor of the menutext_s control is the `MTYPE_PTEXT` type. When `generic.type` is set to this value, the text control again acts like a banner, rendering the text in a larger, proportional font. The main difference with this style is that the control also responds to user input via the keyboard or mouse pointer.

The entire list of required values that will properly initialize a menutext_s control is found in Table 9.3. If you look at the initialization of the main menu's choices starting on line 256 of ui_menu.c, you can see these variables in action.

In the following code snippet, the menutext_s control for the Single Player menu is set up:

Table 9.3 Required Inits for menutext_s

Variable	Value
generic.type	This member is set to either MTYPE_TEXT, MTYPE_BTEXT, or MTYPE_PTEXT.
generic.x	This member sets the control's x location on the screen.
generic.y	This member sets the control's y location on the screen.
generic.flags	QMF_GRAYED is allowed on MTYPE_TEXT and MTYPE_BTEXT. If the type is MTYPE_PTEXT, it can also be QMF_PULSEIFFOCUS, QMF_CENTER_JUSTIFY, and QMF_RIGHT_JUSTIFY.
string	This member contains the text label that is rendered to the left of the control.
style	This member holds bit flags that modify the text alignment and size. Size flags are ignored for MTYPE_BTEXT.
color	This member holds the color of the text that will be rendered. QMF_GRAYED overrides this value.

```
    y = 134;
    s_main.singleplayer.generic.type        = MTYPE_PTEXT;
    s_main.singleplayer.generic.flags       =
QMF_CENTER_JUSTIFY|QMF_PULSEIFFOCUS;
    s_main.singleplayer.generic.x           = 320;
    s_main.singleplayer.generic.y           = y;
    s_main.singleplayer.generic.id          = ID_SINGLEPLAYER;
    s_main.singleplayer.generic.callback    = Main_MenuEvent;
    s_main.singleplayer.string              = "SINGLE PLAYER";
    s_main.singleplayer.color               = color_red;
    s_main.singleplayer.style               = style;
```

Here, the control is set to be of type MTYPE_PTEXT, which will allow the user to select it with keyboard navigation or by clicking on it with the mouse pointer. The flags specify that the control will pulse if it is currently active, and that the text is to be centered. The x and y location of the control are set to 320 and 134, respectively (note that y is set in

Coloring Without Crayons

Because color is handled frequently throughout the `ui` code, the programmers at ID went ahead and defined some variables to represent the most commonly used colors in the menu system, as is shown by the assignment of the `color_red` variable in the preceding code. You can find all the defined colors starting on line 23 of ui_qmenu.c. Each color is of type vec4_t, which is simply a four-dimensional array holding numerical values that represent the amount of red, green, blue, and transparency in the color you want to render.

To create your own color definitions, simply divide each component of the color's RGB value (which ranges from 0 to 255) by 255. (RGB values for colors can often be determined through the use of graphic-editing tools such as Photoshop, and also by HTML editors, because they use RGB formats to specify colors in Web sites.) For example, the color red has an RGB value of (255,0,0), which translates to (1.0, 0.0, 0.0), while a deep purple (128,0,128) translates to (0.5, 0.0, 0.5). Additionally, you can specify the level of transparency of the text, which makes the color "see-through" when placed on top of other backgrounds. 1.0 is completely opaque, whereas 0.0 is fully transparent.

the first line). The string of text to be displayed is "SINGLE PLAYER," and it will be drawn in red.

Note that the `style` member is set to the value of a local variable, also called `style`. Scrolling up a few lines to 232, you can see the declaration and assignment of this local variable.

```
int         style = UI_CENTER | UI_DROPSHADOW;
```

The `style` of a `menutext_s` has additional bit flags that can be assigned to it to assist in formatting and layout. These flags are listed in Table 9.4.

Table 9.4 Generic Text-Formatting Flags

Name	Description
UI_LEFT	This flag draws the control starting at its x, y location.
UI_CENTER	This flag draws the control so that its center is nearest its x, y location.
UI_RIGHT	This flag draws the control so that it ends at the x, y location.
UI_SMALLFONT	This flag draws the control with a small, fixed-width font, held in SMALLCHAR_WIDTH and SMALLCHAR_HEIGHT (8 × 16).
UI_BIGFONT	This flag draws the control with a medium sized, fixed-width font, held in BIGCHAR_WIDTH and BIGCHAR_HEIGHT (16 × 16).
UI_GIANTFONT	This flag draws the control with a large sized, fixed-width font, held in GIANTCHAR_WIDTH and GIANTCHAR_HEIGHT (32 × 48).
UI_DROPSHADOW	This flag draws a shadow below the text.
UI_BLINK	This flag allows the control to flash on and off. Unlike a pulse, there is no gradual transition between the bright and dark flashes.
UI_PULSE	This flag allows the control to fade in and out.

NOTE

There is also a remaining flag, UI_INVERSE, which has been changed throughout various releases of *Q3* so that it no longer inverts text, but instead reduces brightness.

These extra flags allow generic text controls to be formatted in other ways, but there is a key piece of information to remember: The flags applied to style must match those applied to generic.flags. For example, in the previous snippet that describes the Single Player menu, the generic.flags value contains QMF_CENTER_JUSTIFY, while the style value contains UI_CENTER. If QMF_LEFT_JUSTIFY were to be used with UI_CENTER, some crazy alignment would occur, so it's worth mentioning that keeping consistency between generic.flags and

style will save you hours of headaches trying to align your controls properly.

Breathing Life into a Menu

A menu framework cannot exist by definition alone; a menu interacts with the user, receiving input and turning it into data for *Q3* to interpret. To make a menu come to life, you must give it the ability to handle events that a user will *invoke* by clicking on buttons, typing text, cycling through options, and so forth. Because a user interface is nothing if it doesn't respond to input, you need a way to facilitate input by the user. This is done using a *callback function*. If you'll recall, during the listing of the Single Player control there was a reference to a member called callback (which, coincidentally, was listed in the menucommon_s struct). The callback member is simply a pointer to a function, which tells *Q3* what function will run when a user activates the control.

Let's continue with the Single Player control as an example. Line 261 of ui_menu.c handles this initial assignment:

```
s_main.singleplayer.generic.callback    = Main_MenuEvent;
```

Here, callback is set to use the Main_MenuEvent function when it is activated. Main_MenuEvent, as it happens, is a giant switch statement that hands control of the main menu system to another menu, based on the control that was activated. It does this by looking at the unique identifier (id) of the control that calls Main_MenuEvent. The id variable is also a member of menucommon_s, and each control in the *Q3* UI has a unique combination of an id and a callback. It is perfectly viable to have one control with a specific id have different callback functions; the same is true for one callback function to be called by controls of different id. For the Single Player control, the id is set on line 260.

```
s_main.singleplayer.generic.id              = ID_SINGLEPLAYER;
```

The variable ID_SINGLEPLAYER is declared at the top of ui_menu.c, and has a value of 10. To see what actually happens in Main_MenuEvent, let's take a look at its listing at line 67 of ui_menu.c.

```
void Main_MenuEvent (void* ptr, int event) {
    if( event != QM_ACTIVATED ) {
```

```
        return;
    }

    switch( ((menucommon_s*)ptr)->id ) {
    case ID_SINGLEPLAYER:
        UI_SPLevelMenu();
        break;

    case ID_MULTIPLAYER:
        UI_ArenaServersMenu();
        break;

    case ID_SETUP:
        UI_SetupMenu();
        break;

    case ID_DEMOS:
        UI_DemosMenu();
        break;

    case ID_CINEMATICS:
        UI_CinematicsMenu();
        break;

    case ID_MODS:
        UI_ModsMenu();
        break;

    case ID_TEAMARENA:
        trap_Cvar_Set( "fs_game", "missionpack");
        trap_Cmd_ExecuteText( EXEC_APPEND, "vid_restart;" );
        break;

    case ID_EXIT:
        UI_ConfirmMenu( "EXIT GAME?", NULL, MainMenu_ExitAction );
        break;
    }
}
```

Main_MenuEvent requires the following two parameters to be passed into it:

- **A void pointer.** This is a special type of C pointer that can point to any type of data. The benefit of using a void pointer is that any kind of data can theoretically be passed to this function, albeit with a price. The price is that the value of the data being pointed to cannot be determined simply by dereferencing it with the asterisk (as in *ptr = myvalue). A void pointer must be temporarily converted or *cast* to the data type being pointed to, which means the programmer (you) needs to know what type of data it is. Fortunately, you know what it will be in the ui: menucommon_s, the generic struct that makes up every single control.
- **An integer.** This represents the event that was invoked by the control. All controls in the *Q3* UI have three events: QM_GOTFOCUS, QM_LOSTFOCUS, and QM_ACTIVATED. All three of these variables are defined on line 123 of ui_local.h. The QM_GOTFOCUS and QM_LOSTFOCUS events are self-explanatory; they are invoked when a control is first made active, and when it is skipped by after having been made active, respectively. These events can be handy for building custom menus that cause additional animations, sounds, or other effects when the user visits a control. The main event (if you'll pardon the pun) is QM_ACTIVATED, which is invoked when the control is currently taking input from the user.

At the beginning of this function, the event variable is checked to see whether it is not QM_ACTIVATED, exiting the function if this evaluation is true. Then the switch block begins, based on the id of the control that called it. Notice the value of the control's id being accessed by dereferencing the pointer (adding the * to ptr), after casting the pointer to the data type menucommon_s. Next, various case statements are set, based on the value that was found in the id. Because you know that the Single Player control's id is ID_SINGLEPLAYER, the UI_SPLevelMenu function takes over.

Tweaking *Q3*

In order to start putting the menu framework to good use, you will now start creating your own framework from scratch. The new menu

you'll build will allow a user to tweak various settings in *Q3* that otherwise need to be changed by direct manipulation through the console. These include settings for such features as shadow quality, allowing the view to be in third person, adjusting the player's field of view, and typing in a text string to represent the player's gender. Each of these options can be implemented using one of the seven types of controls available, so this is a good opportunity to get better acquainted with them.

In order to add a new menu, the first step is to set aside a new ID for the menu control that will be added to the main menu. Start by opening ui_menu.c, and looking at the defines near the top of the page. You should see something like the following:

```
#define ID_SINGLEPLAYER         10
#define ID_MULTIPLAYER          11
#define ID_SETUP                12
#define ID_DEMOS                13
#define ID_CINEMATICS           14
#define ID_TEAMARENA            15
#define ID_MODS                 16
#define ID_EXIT                 17
```

These variables represent the various choices of the main menu (see Figure 9.2), which are Single Player, Multiplayer, Setup, Demos, Cinematics, Team Arena (if it's installed), Mods, and Exit.

Go ahead and add an ID for the new Tweaks menu you will build. Bump the `ID_EXIT` value up by one (`18`) and slide a define for `ID_TWEAKS` in at `17`, so that the defines look like this near the end:

```
#define ID_MODS                 16
#define ID_TWEAKS               17 // our new tweaks menu
#define ID_EXIT                 18
```

You will now be able to refer to the menu by its unique identifier. Next, you want to slip a new menutext_s into the list of current menutext_s controls that form the members of mainmenu_t (which you looked at earlier in this chapter). Scroll down to line 39 and squeeze a new menutext_s control declaration in between `mods` and `exit`, like so:

```
    menutext_s      mods;
    menutext_s      tweaks; // our new tweaks menu control
    menutext_s      exit;
```

Figure 9.2 *The Q3 main menu*

Perfect. You now have a control that will allow the user to enter your new menu. The next item on your to-do list is to set up the control's default values for its various members. This includes the required initializations of the members held in the menucommon_s struct (which is the `generic` variable), and any specific values that are required for the control in question. In the case of the menutext_s, those will be `string`, `color`, and `style`.

Setting the Stage for a Menu

Scroll down to line 341; this should put you hip-deep in the middle of `UI_MainMenu`, the function that sets up all the necessary data for the controls in the main menu, and then activates the menu, bringing it up for the user to access. Because every menu follows in the footsteps of the main menu, a lot can be learned from how it works. In a nutshell, `UI_MainMenu` executes in the following manner:

1. It clears the memory in the variable that will hold the menu.
2. It caches any images, sounds, or other necessary data.

3. It initializes the menu.
4. It initializes all controls used in the menu.
5. It adds controls to the menu.
6. It pushes the menu to the screen.

Every menu in the UI is created in this manner, so let's see what the specifics are to achieve each step. First, clearing the memory of the variable that will hold the menu is done on line 252 of ui_menu.c.

```
memset( &s_main, 0 ,sizeof(mainmenu_t) );
```

You've worked with `memset` a few times already; it is a C function that allows you to set all the memory in a variable's space to a certain value, which is most commonly `0`. This effect clears the variable of any unnecessary values that may be lurking. Notice that the third parameter of `memset` is `sizeof(mainmenu_t)`, which you should recognize as the struct that `s_main` is declared as.

The second step is to cache images and sounds that will be used in the menu. This is performed on the very next line, with a call to `MainMenu_Cache`. It just so happens that `MainMenu_Cache` is in the same file, up near line 120:

```
void MainMenu_Cache( void ) {
    s_main.bannerModel = trap_R_RegisterModel( MAIN_BANNER_MODEL );
}
```

Because the only real graphical data that needs to be cached is the animated "Quake III Arena" text across the top of the screen, the content of this function is a mere one-liner. You should recall from our discussion of mainmenu_t that `bannerModel` is a qhandle_t, which you have used before in creating references to sounds, icons, shaders, and the like.

After the caching is completed, the next step is to initialize the menu. As mentioned earlier in this chapter, you will want to make sure the three main members of menuframework_s are set up appropriately. Line 256 demonstrates this.

```
s_main.menu.draw = Main_MenuDraw;
s_main.menu.fullscreen = qtrue;
s_main.menu.wrapAround = qtrue;
```

The draw property is set to run the function Main_MenuDraw, which handles the custom logo, assigning coordinate points locations for rendering, and ending with a call to Menu_Draw.

TIP

All menus are drawn as if being rendered to a 640 × 480 resolution. If you happen to be viewing *Q3* in a higher resolution, the ui code will automatically adjust positions and resize controls for you.

The fullscreen member is set to qtrue, meaning it will take full control of *Q3*, pausing any game currently in progress on the client. wrapAround is initialized to qtrue as well, meaning the active control will cycle continuously if the user continues to move down past the last choice (which would be Exit).

After the menu is initialized, all of its controls must suffer the same fate. Starting on line 261, each control has a chunk of code dedicated to setting up all the various values that are required for the control to come to life in the menu. Jump down to line 341, and add your new control here, just after the Mods control is finished being initialized.

```
    s_main.mods.style                       = style;

    // setup the new menu
    y += MAIN_MENU_VERTICAL_SPACING;
    s_main.tweaks.generic.type              = MTYPE_PTEXT;
    s_main.tweaks.generic.flags             =
QMF_CENTER_JUSTIFY|QMF_PULSEIFFOCUS;
    s_main.tweaks.generic.x                 = 320;
    s_main.tweaks.generic.y                 = y;
    s_main.tweaks.generic.id                = ID_TWEAKS;
    s_main.tweaks.generic.callback          = Main_MenuEvent;
    s_main.tweaks.string                    = "TWEAKS";
    s_main.tweaks.color                     = color_red;
    s_main.tweaks.style                     = style;
```

Now, a new control will be used to access your Tweaks menu. It uses the current y variable, plus an additional MAIN_MENU_VERTICAL_SPACING adjustment, to keep it in sync with the spacing used for the previous controls. The generic members are set next: generic.type is set to MTYPE_PTEXT, generic.flags receives QMF_CENTER_JUSTIFY and

QMF_PULSEIFFOCUS, generic.x gets 320, generic.y gets the same local y value you set earlier, and id gets the ID_TWEAKS variable you defined at the beginning of this section. Then, it's on to the specific values of a menutext_s control: string gets the value "TWEAKS", which will be its label in the menu, color is assigned the color_red variable, and style assumes the same style variable applied to all controls in this menu, namely UI_CENTER and UI_DROPSHADOW (set on line 238).

But what about that sneaky callback function? Here, it is set to Main_MenuEvent, just like all the others. I have a feeling that Main_MenuEvent isn't quite ready to handle the new Tweak menu yet, though. Scroll up to around line 104, where power over the UI is passed to various menus based on the control that is clicked, and under the case for ID_TEAMARENA, make the following additions:

```
case ID_TWEAKS:
    UI_TweaksMenu(); // handing control off to tweaks menu
    break;
```

Now the Main_MenuEvent knows how to handle your new menu. It will check to see if the ID of the control that was activated was ID_TWEAKS, and if so, will pass control to UI_TweaksMenu (a function you will write later).

Pushing a Menu Will Only Make It Mad

The hard stuff is over for this menu. The last two items required to make it come to life are to add all the initialized controls to the menu and then push the menu to the screen so that the user may access it. The controls are added with a series of calls to a function called Menu_AddItem, which takes two parameters, a menuframework_s, and a control. Looking down at line 364, you can see Menu_AddItem being called multiple times:

```
Menu_AddItem( &s_main.menu,     &s_main.singleplayer );
Menu_AddItem( &s_main.menu,     &s_main.multiplayer );
Menu_AddItem( &s_main.menu,     &s_main.setup );
Menu_AddItem( &s_main.menu,     &s_main.demos );
Menu_AddItem( &s_main.menu,     &s_main.cinematics );
if (teamArena) {
    Menu_AddItem( &s_main.menu,     &s_main.teamArena );
```

```
    }
    Menu_AddItem( &s_main.menu,     &s_main.mods );
    Menu_AddItem( &s_main.menu,     &s_main.exit );
```

One by one, each control is added to the menu by specifying the `menu` member of `s_main`, and then the control that is currently being added. Go ahead and use this format to add your new control in, right between `mods` and `exit`:

```
    Menu_AddItem( &s_main.menu, &s_main.mods );
    Menu_AddItem( &s_main.menu, &s_main.tweaks ); // adding the new
control!
    Menu_AddItem( &s_main.menu, &s_main.exit );
```

Now for the final step. This is very complicated, so pay extremely close attention.

```
UI_PushMenu ( &s_main.menu );
```

OK, so I was being a little sarcastic. The menu is brought to the screen for user accessibility by simply calling the function `UI_PushMenu`, passing in the menuframework_s variable that refers to the menu that needs to be active: `s_main`. *Q3* handles all the rest for you. Pretty slick, eh? Now that the control to access the new Tweaks menu is ready to go, you need to build the actual menu . . . and seeing as how you just stepped through all the requirements to create a menu, you should be raring to go.

Building a New UI Menu

In this section, you will look at what it takes to create a user-interface menu from scratch by building a new menu framework, laying in custom controls, and creating a *callback handler* function to handle any user interaction that will take place in the menu. After the new menu is built, you'll have a solid structure on which to base further additions, such as new controls or updates to the layout.

Starting ui_tweaks.c

Each of the menus in *Q3* has its own setup file; the main menu's code resides in ui_menu.c, the preferences are held in ui_preferences.c, the sound configuration menu has its parts in ui_sound.c, and so on.

Because you are adding a brand-new menu, you should place it within a new file as well. To do this, click the toolbar button in Visual Studio that looks like a piece of paper with a corner folded over. This is the New Text File icon, shown in Figure 9.3.

You will want to save this new text file right away so you can add it to the existing `ui` code (and remember, this means you will add it to the `q3_ui` project). Go ahead and save the file by pressing Ctrl+S, or by clicking the Save toolbar button (the one with a disk on it), shown in Figure 9.4.

You are prompted to name the new file, and to specify where you want to save it. Type **ui_tweaks.c** and save it in the /quake3/code/q3_ui/ folder (if you aren't currently in that folder, use the Save In drop-down list to select that folder path). Then, slick the Save button to commit the new file to your hard drive. Excellent! Now all you need to do is add the new file to the `q3_ui` project. You can add a new file to the project simply by right-clicking the Source Files folder, and selecting Add Files to Folder from the pop-up menu that appears, as shown in Figure 9.5.

A dialog box should open, allowing you to look through folders for files to be added to the project. Find your new ui_tweaks.c file in the /quake3/code/q3_ui/ folder, and add it now. When this task is

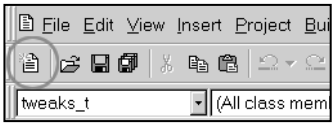

Figure 9.3 *The New Text File button*

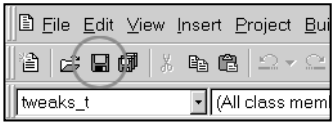

Figure 9.4 *The Save button*

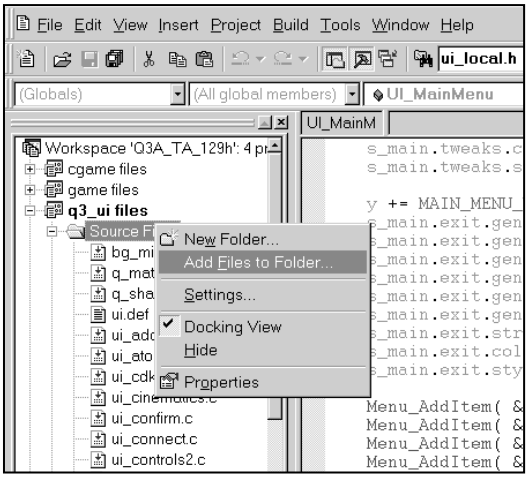

Figure 9.5 *Right-clicking the Source Files folder*

completed, the file should be listed in the Source Files folder, near the bottom, next to ui_teamorders.c and ui_video.c. Any code that exists in this file will now be compiled along with the rest of the `ui` code when the final DLL is being built.

Now that you have a new menu file, let's start dropping some code into it. First, you will want to include the ui_locals.h header, because it includes variables and declarations for all common UI functionality. The first few lines of your new ui_tweaks.c file should read as follows:

```
//
// ui_tweaks.c
//
#include "ui_local.h"
```

> **TIP**
>
> **As you can see, I've gone ahead and added a C comment at the top, letting everyone know what the name of this file happens to be. There's nothing wrong with being courteous when coding, and comments always help other programmers understand what you were thinking when you created your code.**

Next, you will set up some defines that will represent the first controls and graphics added to this menu. You'll start with the bare necessities first, and add extras later. Most of the menus in the *Q3* user interface have a curved left and right bracket image that surround the menu choices, so to keep consistency, you'll use them as well. Also, you need to add a button to allow the user to back out of your menu if he wants to navigate to another menu. The Back button consists of two images: a bright version, for when the user's mouse is hovering over it, and a dim version, for when it sits idle on the screen. The four defines for the brackets and the back button images go next, after the `#include "ui_local.h"` line:

```
#define ART_BACK0               "menu/art/back_0"
#define ART_BACK1               "menu/art/back_1"
#define ART_FRAMEL              "menu/art/frame2_l"
#define ART_FRAMER              "menu/art/frame1_r"
```

`ART_BACK0` and `ART_BACK1` are the references to the Back button images, while `ART_FRAMEL` and `ART_FRAMER` reference the left and right bracket images.

TIP

When referencing images in a file hierarchy such as /menu/art/, if no file extension is specified, then the format TGA is assumed. So, in the code above, `back_0.tga`**,** `back_1.tga`**,** `frame2_l.tga` **and** `frame1_r.tga` **are the actual names of the files used. You are also free to specify a different file extension if you want to use a file type other than TGA, such as menu/art/back_0.jpg.**

Defining the Menu Struct

For the initial Tweaks menu, you'll allow the user to change a client Cvar called `cg_thirdPerson`, which determines the point of view of the player in *Q3*. By default, this variable is off, with a value of `0`. If it is set to `1`, the player's view shifts so that the player's model in the game is visible, in what is often referred to as a chase-cam view (shown in Figure 9.6).

Figure 9.6 *Third-person view enabled in* Q3

Because there are only two values that the cg_thirdPerson variable can be (on or off), the perfect control for the job is the radio-button control, *menuradiobutton_s*. The radio button (or "option" button), if you'll recall, is a round dot that is either filled to represent being selected or "on," or cleared out to represent being deselected or "off." Now that you have two identifiable controls (the third-person radio button and the Back button), add the defines for those two controls next, after ART_FRAMER, so that they read like so:

```
#define ID_BACK                    10
#define ID_THIRDPERSON             11
```

Good work, the IDs are in place. The next task is to lay out a new struct that will house the Tweaks menu variable. This will be the declaration of the tweaks_t struct, which follows the defines listed previously:

```
typedef struct {
    menuframework_s     menu;

    menubitmap_s        framel;
    menubitmap_s        framer;

    menutext_s          banner;
    menuradiobutton_s   thirdPerson;
    menubitmap_s        back;
} tweaks_t;
static tweaks_t    s_tweaks;
```

The first member of a menu struct must always be the menuframework_s, so it is first in this list of members. Two menubitmap_s controls are used to house the left and right bracket images. The title of the menu, "Tweaks," is to be held in a menutext_s control. Then, the thirdPerson variable is declared to be of type menuradiobutton_s, as discussed earlier. Finally, one additional menubitmap_s control is added to reference the Back button. Once the struct is complete, a global static variable is declared to be of type tweaks_t, called s_tweaks.

Getting a Handle on Menu Events

The next function that is needed is the event-handling method. When the user manipulates a control, you'll want to have the proper command called within *Q3* to respond. Currently, only two controls will

ever be accessed: the Back button and the radio button allowing the user to set his cg_thirdPerson preference. Go ahead and add the following function after your variable definition for s_tweaks:

```
/*
===============
UI_Tweaks_MenuEvent
===============
*/
static void UI_Tweaks_MenuEvent( void *ptr, int event ) {
    if( event != QM_ACTIVATED ) {
        return;
    }

    switch ( ((menucommon_s*)ptr)->id ) {

    case ID_THIRDPERSON:
        trap_Cvar_SetValue( "cg_thirdPerson",
s_tweaks.thirdPerson.curvalue );
        break;

    case ID_BACK:
        UI_PopMenu();
        break;
    }
}
```

This is the body of the function UI_Tweaks_MenuEvent, which, like Main_MenuEvent, will take a void pointer and an integer, representing the event invoked by the control. A sanity check on the event variable is performed, to confirm that it is indeed QM_ACTIVATED. Then the switch block begins, looking at the value of the void pointer ptr (which is cast to a menucommon_s data type before being dereferenced). The first case is the ID_THIRDPERSON control ID. If the control is activated, a call to trap_Cvar_SetValue is made, assigning the current value of the radio button (held in the curvalue member) to the Cvar cg_thirdPerson. You'll see more of curvalue and the rest of the menuradiobutton_s control in a bit.

The only other control you have is identified by ID_BACK, the Back button. If it is clicked, you simply remove the Tweaks menu from view, which will place the user at the previous menu (in this case, the main menu). This is done by a simple call to UI_PopMenu.

Initializing the Menu Controls

The function that handles the initialization of the menu and its controls is a doozy, so I'll take it step-by-step. Go ahead and add the lines that I walk through in this section, following all the previous code you've added to ui_tweaks.c. The function, `UI_Tweaks_MenuInit`, will start as follows:

```
/*
===============
UI_Tweaks_MenuInit
===============
*/
static void UI_Tweaks_MenuInit( void ) {
    UI_TweaksMenu_Cache();

    memset( &s_tweaks, 0 ,sizeof(tweaks_t) );

    s_tweaks.menu.wrapAround = qtrue;
    s_tweaks.menu.fullscreen = qtrue;
```

The function opens simply by making a call to `UI_TweaksMenu_Cache`, a function that you will write later, handling the setup of the various graphical objects in this menu. Next, the memory of the `s_tweaks` variable is cleared with a call to `memset`. Following that, the `wrapAround` property of the `s_tweaks.menu` is set to `qtrue`, and the `fullscreen` property is also set to `qtrue`.

Next, the title of the menu that will be rendered as a banner across the top of the screen is initialized, with the following code:

```
    s_tweaks.banner.generic.type          = MTYPE_BTEXT;
    s_tweaks.banner.generic.x             = 320;
    s_tweaks.banner.generic.y             = 16;
    s_tweaks.banner.string                = "TWEAKS";
    s_tweaks.banner.style                 = UI_CENTER;
```

This code should be fairly straightforward to you by now; `MTYPE_BTEXT` means the text will be large and in a banner-style font, the x and y location will be 320 × 16 on the screen, the text will read "TWEAKS", and the style will be `UI_CENTER`, which will be centered at its x, y location.

The next control will be the menuradiobutton_s control; take a peek at Table 9.5 to see what its required initializations are.

Armed with the information in Table 9.5, you can next initialize the radio button with the following code:

```
    s_tweaks.thirdPerson.generic.type         = MTYPE_RADIOBUTTON;
    s_tweaks.thirdPerson.generic.flags        = QMF_PULSEIFFOCUS |
QMF_SMALLFONT;
    s_tweaks.thirdPerson.generic.x            = 320;
    s_tweaks.thirdPerson.generic.y            = 130;
    s_tweaks.thirdPerson.generic.name         = "Use Third-Person View";
    s_tweaks.thirdPerson.generic.id           = ID_THIRDPERSON;
    s_tweaks.thirdPerson.generic.callback     = UI_Tweaks_MenuEvent;
    s_tweaks.thirdPerson.curvalue             = trap_Cvar_VariableValue(
"cg_thirdPerson" ) != 0;
```

Here you see the type is `MTYPE_RADIOBUTTON`, and the formatting flags are `QMF_PULSEIFFOCUS` and `QMF_SMALLFONT`. The x and y location on the page will be 320 × 130 and the text displayed next to the control will be "Use Third-Person View." The ID is set (of course) because this control will need to be identified when it is activated, so its `generic.id` member is set to `ID_THIRDPERSON`. The callback function that will handle the button's event is `UI_Tweaks_MenuEvent`, the function you wrote earlier. Finally, the `curvalue` (whether the button is on or off) is set to the value of `trap_Cvar_VariableValue`, a system-call function that returns the value of a Cvar. In this particular instance, if the value of

Table 9.5 Required Inits for menuradiobutton_s

Variable	Value
`generic.type`	This member is set to `MTYPE_RADIOBUTTON`.
`generic.x`	This member sets the control's x location on the screen.
`generic.y`	This member sets the control's y location on the screen.
`generic.name`	This member holds the text display to the left of the button.
`curvalue`	This member equals the current value of the button: `1` if the button is "on" and `0` if the button is "off."

the function return does not equal 0, curvalue will receive a value of 1; otherwise, it will receive a 0.

The final three controls are bitmaps. Two are static, meaning they just sit and look pretty; they do not animate or respond to user input in anyway, those being the left and right bracket graphics. The third control is the Back button image, and it will interact with the user. All three are of control data type menubitmap_s, which has its most important members listed in Table 9.6.

Table 9.6 Required Inits for menubitmap_s

Variable	Value
generic.type	This member is set to MTYPE_BITMAP.
generic.x	This member sets the control's x location on the screen.
generic.y	This member sets the control's y location on the screen.
generic.flags	If the bitmap is static and non-interactive, this member is set to QMF_INACTIVE; otherwise, standard formatting flags can be applied.
generic.name	This member is assigned to the image filename and path to load the image. Setting this will automatically handle setting shader as well.
shader	This member is assigned to the filename and path of the shader, if needed.
errorpic	This member is assigned to the image filename and path to load if the main image in generic.name cannot be found or loaded.
focuspic	This member is assigned to the image filename and path to load if the control is active by the keyboard or mouse pointer.
focusshader	This member is assigned to the filename and path of the shader to be used when the control is active, if needed.
focuscolor	This member specifies the color of the image when it is made active.
width	This member specifies the width of the image.
height	This member specifies the height of the image.

Because using an image in a menu requires the most flexibility, there are a good number of members that can be assigned values, as the listing denotes. Luckily, you are going to be using as many of the defaults as necessary. Go ahead and add the following code to initialize the three remaining controls:

```
s_tweaks.framel.generic.type            = MTYPE_BITMAP;
s_tweaks.framel.generic.name            = ART_FRAMEL;
s_tweaks.framel.generic.flags           = QMF_INACTIVE;
s_tweaks.framel.generic.x               = 0;
s_tweaks.framel.generic.y               = 78;
s_tweaks.framel.width                   = 256;
s_tweaks.framel.height                  = 329;

s_tweaks.framer.generic.type            = MTYPE_BITMAP;
s_tweaks.framer.generic.name            = ART_FRAMER;
s_tweaks.framer.generic.flags           = QMF_INACTIVE;
s_tweaks.framer.generic.x               = 376;
s_tweaks.framer.generic.y               = 76;
s_tweaks.framer.width                   = 256;
s_tweaks.framer.height                  = 334;

s_tweaks.back.generic.type              = MTYPE_BITMAP;
s_tweaks.back.generic.name              = ART_BACK0;
s_tweaks.back.generic.flags             = QMF_LEFT_JUSTIFY|
                                          QMF_PULSEIFFOCUS;
s_tweaks.back.generic.id                = ID_BACK;
s_tweaks.back.generic.callback          = UI_Tweaks_MenuEvent;
s_tweaks.back.generic.x                 = 0;
s_tweaks.back.generic.y                 = 480-64;
s_tweaks.back.width                     = 128;
s_tweaks.back.height                    = 64;
s_tweaks.back.focuspic                  = ART_BACK1;
```

Notice that the main difference between the controls here is that the first two have their `generic.flags` members set to `QMF_INACTIVE`. Because they are seen as inactive controls in the `ui` code, they do not require the `id` or `callback` assignments that active controls do. The third control will definitely be interactive, so its `generic.flags` has standard formatting flags assigned to it—`QMF_LEFT_JUSTIFY` and `QMF_PULSEIFFOCUS`. The `id` is set to `ID_BACK`, and the `callback` function is set to

UI_Tweaks_MenuEvent. It also has a focuspic member set to ART_BACK1, the image to be drawn over top of the main image, ART_BACK0, when the control has focus from the user. Note also that all three controls have their appropriate width and height members specified.

Phew, you're almost done with this init function! The last step required in initialization is to add all these controls to the s_tweaks.menu variable, so add the following code at the very end of your UI_Tweaks_MenuInit function:

```
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.banner );
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.thirdPerson );
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.framel );
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.framer );
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.back );
}
```

Perfection! Now you have a completed initialization function. This menu is almost ready to go.

The Cache and Push

Your remaining tasks are simple: Write a function to handle the caching of the graphical data and a function that will push the menu to the screen, when active. Let's start with the caching function, which you'll recall from an earlier code reference will be named UI_TweaksMenu_Cache:

```
/*
=================
UI_TweaksMenu_Cache
=================
*/
void UI_TweaksMenu_Cache( void ) {
    trap_R_RegisterShaderNoMip( ART_BACK0 );
    trap_R_RegisterShaderNoMip( ART_BACK1 );
    trap_R_RegisterShaderNoMip( ART_FRAMEL );
    trap_R_RegisterShaderNoMip( ART_FRAMER );
}
```

The UI_TweaksMenu_Cache function is fast and simple. It passes the four defined image variables (the two Back buttons and the two bracket images) to the system-call function trap_R_RegisterShaderNoMip, which

you should remember from Chapter 6. This function is called at the start of your giant initialization function, UI_Tweaks_MenuInit.

I can see the end in sight! The last function to write, called UI_TweaksMenu (which you'll recall is the menu to which that control is handed off by the Main_MenuEvent function back in ui_menu.c), will push the menu to the screen. Quick! Slap this code in at the end of the ui_tweaks.c file, and pronto!

```
/*
===============
UI_TweaksMenu
===============
*/
void UI_TweaksMenu( void ) {
        UI_Tweaks_MenuInit();
        UI_PushMenu( &s_tweaks.menu );
}
```

The role of this function is to first call the giant initialization function, UI_Tweaks_MenuInit, and then to push the menu to the screen with UI_PushMenu.

You've crossed the finish line! You now have all the necessary code in place to handle a brand new menu.

Cleaning Up

Before the DLL is built, you must take care of a few items to make the C compiler happy. For starters, two of your functions must be prototyped. UI_TweaksMenu_Cache is called from UI_Tweaks_MenuInit before it is defined in the file ui_tweaks.c. Additionally, UI_TweaksMenu itself is called from another file, ui_menu.c. So, to declare both functions ahead of time, open ui_local.h, scroll down to line 310 (right after the prototypes for InGame_Cache and UI_InGameMenu), and enter the following lines of code:

```
//
// ui_tweaks.c
//
extern void UI_TweaksMenu_Cache( void );
extern void UI_TweaksMenu( void );
```

Now when you attempt to build your new DLL, the compiler will know how to handle these functions.

The moment of truth is upon you. Go ahead and select Batch Build from Visual Studio's Build menu and uncheck everything except the following:

```
q3_ui - Win32 Release
```

That's the one you want to build! Click the Build button, and let her rip! If all goes well, your Build Dialog result (the window near the bottom of the IDE that display compile information) should finish with the following:

```
ui_teamorders.c
ui_tweaks.c
ui_video.c
Linking...
   Creating library Release/uix86.lib and object Release/uix86.exp
Creating browse info file...

uix86.dll - 0 error(s), 0 warning(s)
```

Can you see your new ui_tweaks.c in that list? There it is! Now, browse over to your /quake3/code/Release/ folder and you should see a uix86.dll. Go ahead and drop it in your MyMod folder, and launch *Q3*, remembering to set `fs_game` to `MyMod` and `sv_pure` to `0`. You should see the Tweaks menu somewhere in your list, as shown in Figure 9.7.

Try going into it by clicking on it with the mouse or selecting it with the keyboard arrow controls. You should be able to enter the new Tweaks menu and select the new Use Third-Person View option (see Figure 9.8).

You now have another notch in your belt—you've successfully completed the necessary steps to create a menu framework and add it to

TIP

A console command called `ui_cache` automatically runs all the UI-related caching functions in sequence. If you want, you can add your menu's caching function to this list as well. Open the file ui_atoms.c, scroll to line 881 to the function `UI_Cache_f`, and insert `UI_TweaksMenu_Cache` anywhere you wish. It will then be included in the global caching execution when the `ui_cache` command is typed into the console. I've gone ahead and done it for you in the code for this chapter on the CD.

Figure 9.7 *The all-new* Q3 *main menubject*

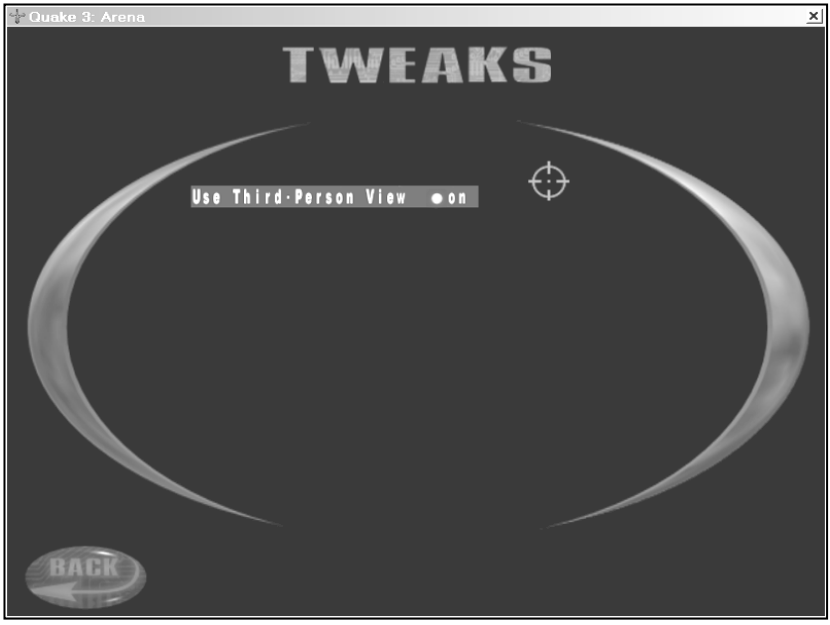

Figure 9.8 *The Tweaks menu*

the existing menu system with the *Q3* user interface. From there, you've added a new control to that menu, allowing the user to manipulate a Cvar, without having to bother with remembering the name of the variable in question. From here on in, things just get easier with the `ui` code.

Working with More Controls

Now that you've had a chance to play around with the menu system in the `ui` code and have gotten familiar with the ins and outs of creating a menu framework, adding controls, and integrating the new menu with existing menus, let's take some time to investigate the remaining controls and what functionality they can offer you in the quest to build the perfect interface to your next exciting project. I went over the basic controls menubitmap_s, menuradiobutton_s, and menutext_s in the previous section. What follows is a look at menufield_s, a control for text input, and menuslider_s, a control for allowing a degree of value via a slider. Finally, you'll look at menulist_s, a control that lets a user cycle through a series of choices.

menufield_s of Dreams

If you want to allow players to type free-form text into your interface, *menufield_s* is the control you want. This control is used quite frequently throughout the `ui` code for such functions as allowing the player to name his online character and setting server-related information, like the name of the server, the time limit, and the frag limit. It is also used for specifying connection data, such as the IP address or host name of the online server to which the player wishes to connect, as well as the port.

The menufield_s control is nothing magical, nor is it difficult to understand. It is simply rendered on the screen as a single-line text box that can receive input through the typing of any character on the keyboard. It typically has a fixed width, which is dictated by the designer of the interface (you), and it also has an internal maximum number of characters it can hold. If the user types more characters into a menufield_s control than what can be physically shown by the

control, the characters scroll to the left automatically, to indicate to the user that more characters are being accepted. The guts of the menufield_s control look like this, as seen on line 170 of ui_local.h:

```
typedef struct
{
    menucommon_s generic;
    mfield_t field;
} menufield_s;
```

Like all controls, the first member is a `generic` variable of data type menucommon_s. The only other member is a variable called `field`, of type *mfield_t*. The declaration of the mfield_t struct takes place directly above the declaration menufield_s, and it reads as follows:

```
typedef struct {
    int         cursor;
    int         scroll;
    int         widthInChars;
    char        buffer[MAX_EDIT_LINE];
    int         maxchars;
} mfield_t;
```

Three of these members are specific to what you will use when you place the control in a menu. The `widthInChars` member is an integer that represents the physical width of the control, as it is drawn to the screen. This is the value you will tweak to change the size of the textbox when it is placed in your menu. The `buffer` member is a char array, which C programmers should recognize as a standard way of storing text strings. The array's size or *upper limit* is set to `MAX_EDIT_LINE`, a defined variable equal to 256. Although the size of the array is capped at 256, the variable `maxchars` is the value that the control uses to physically limit the control's maximum number of characters to be stored in the textbox. So if you want a menufield_s to allow only 20 characters to be typed, you can set `maxchars` to `20` and the user will not be allowed type any more than that into the control. You can also skip setting `maxchars`; the default, `MAX_EDIT_LINE` (256) will be used in its place.

As with the previous controls, there are a certain number of required initializations that must take place in order to properly handle a menufield_s control. Table 9.7 lists these required initializations.

Table 9.7 Required Inits for menufield_s

Variable	Value
`generic.type`	This member is assigned a value of `MTYPE_FIELD`.
`generic.x`	This member sets the control's x location on the screen.
`generic.y`	This member sets the control's y location on the screen.
`generic.name`	If a string is assigned to this variable, it will be placed to the left of the control when rendered to the screen; the physical textbox will remain at its x, y location, regardless.
`field.widthInChars`	This member holds the physical character width displayed by the control.
`field.buffer`	This member holds a proper zero-terminated char array string.
`field.maxchars`	This member holds the maximum number of characters the control can accept.

See, I told you that there was nothing complicated here. Go ahead and add a menufield_s control to your existing menu. Another Cvar you are free to mess with is the `sex` Cvar, which holds the gender of the player. Typically, the `sex` Cvar isn't used all that much. It contains a standard string for a value (typically `"Male"` or `"Female"`), which means you could easily modify it with a menufield_s control.

First, you will want to set aside a new unique identifier for the new control. That takes place back in ui_tweaks.c, way up at line 12. Right after the defines of `ID_BACK` and `ID_THIRDPERSON`, add a new define, `ID_SEX`:

```
#define ID_BACK                  10
#define ID_THIRDPERSON           11
#define ID_SEX                   12
```

The new textbox will definitely be accepting input from the user, so it will need an ID assigned to it. This define will set the stage for that assignment.

Next, because you're adding a new control to the Tweaks menu, you will need to set aside a place for it within the tweaks_t struct (which defines your `s_tweaks` menu variable). On line 14, where the tweaks_t struct is declared, add the menufield_s control after the `thirdPerson` variable, and call the new control `sex`. The amended tweaks_t struct should read as follows:

```
typedef struct {
    menuframework_s         menu;

    menubitmap_s            framel;
    menubitmap_s            framer;

    menutext_s              banner;
    menuradiobutton_s       thirdPerson;
    menufield_s             sex;
    menubitmap_s            back;
} tweaks_t;
```

Now that you have a place for the menufield_s control, you'll need to properly initialize it. As memory serves, the initializations for all the controls are wrapped up in `UI_Tweaks_MenuInit`. Scroll down to line 77, where the setup of the radio button for `thirdPerson` ends, and add the following lines of code:

```
    s_tweaks.sex.generic.type               = MTYPE_FIELD;
    s_tweaks.sex.generic.flags              = QMF_SMALLFONT;
    s_tweaks.sex.generic.x                  = 320;
    s_tweaks.sex.generic.y                  = 150;
    s_tweaks.sex.generic.name               = "Gender";
    s_tweaks.sex.generic.id                 = ID_SEX;
    s_tweaks.sex.field.widthInChars         = 18;
    s_tweaks.sex.field.maxchars             = 30;
```

For this particular control, you start with a `generic.type` of `MTYPE_FIELD`, and a `generic.flags` of `QMF_SMALLFONT`. The x and y location is 320 × 150, slightly lower on the screen than the previous control. The label of the field is "Gender" (just to be politically correct),

which is held in generic.name. As for the generic.id of the control, it is set to ID_SEX. The actual size of the textbox that will be accessible to the user will be 18 characters wide. This value is stored in widthInChars. The maxchars will be 30—ample room to hold one word.

Notice a surprisingly vacant member initialization from this set: the assignment of the generic.callback member. If the control is accessed, you certainly want it to be able to set a value that is typed in the textbox to a variable, but how can you do that without a callback function to handle any events the control might invoke? You do this using a new technique: redirecting the callback function that is invoked by the menu when a keypress occurs.

Trapping the Keyboard Red-Handed

Sometimes it isn't necessary for a control to call an update function every single time it changes. The menufield_s control is the perfect example; if you were typing a 256-character string into the control, would you really need the overhead of a function being called every single keypress? Chances are, those extra calls are really unneeded (and any C programmer will tell you that the more processor usage you can avoid, the better). A better method would be to let the control remain idle while characters are entered into it, and then call a final update function when the control is finished being used.

The menuframework_s struct, held in s_team.menu contains a member called key, which is a pointer to a function, much in the same way that think is within a gentity_t, or the callback function is within menucommon_s. This particular function is invoked whenever a keypress is detected within the menu currently being accessed. It could be any key: a letter, a number, the Enter key, the Esc key, or any other valid key you see on the keyboard. By default, the key function simply returns the key that was pressed to the calling function, so that appropriate steps can be taken by the function that called it. You can, however, create a function and point your menu's key member to it, thereby forcing the new function to be called whenever a keypress is detected.

TIP

The default key function is Menu_DefaultKey, and you can read its definition in ui_qmenu.c, way down at line 1563.

Let's go ahead and write a new function that will trap a keypress from the Tweaks menu and hold the information necessary to call an update to your new menufield_s control. Above your definition of UI_Tweaks_MenuEvent on line 51, scroll up a few lines and add the following function:

```
/*
=================
TweaksSettings_MenuKey
=================
*/
static sfxHandle_t TweaksSettings_MenuKey( int key ) {
    if( key == K_MOUSE2 || key == K_ESCAPE ) {
        TweaksSettings_SaveChanges();
    }
    return Menu_DefaultKey( &s_tweaks.menu, key );
}
```

Because the default key function of a menu is Menu_DefaultKey, and that function returns a variable of type sfxHandle_t, your new key handler must also return that, as this definition of TweaksSettings_MenuKey shows. This function is really a no-brainer; the function requires an integer called key (which will hold the value of the key that was pressed) and simply checks to see if that key matches one of two predefined variables, K_MOUSE2, for the second mouse button, or K_ESCAPE, for the Esc key. If either of those two keys is trapped, the TweakSettings_SaveChanges function is called, and then, TweaksSettings_MenuKey exits properly by returning the value from Menu_DefaultKey. Note that the final call to Menu_DefaultKey passes in your s_tweaks.menu variable, along with the trapped keypress held in key.

Now that you have a function trapping keys, let's write the function to save the value currently stored in the menufield_s control. Above the

TIP

Every single key on the keyboard has an appropriate variable declared for it, like K_MOUSE2 and K_ESCAPE. You can find the entire listing in keycodes.h, starting on line 12. There are also variables defined for joystick buttons as well, all falling into a declaration of the *keyNum_t* enum.

TweaksSettings_MenuKey function, add the following function definition for TweaksSettings_SaveChanges:

```
/*
=========================
TweaksSettings_SaveChanges
=========================
*/
static void TweaksSettings_SaveChanges( void ) {
        trap_Cvar_Set( "sex", s_tweaks.sex.field.buffer );
}
```

This straightforward function makes a call to trap_Cvar_set (which you should recognize as a system-call function), setting the Cvar sex to the value currently held in s_tweaks.sex.field.buffer. Now you have a function that will commit the changes typed into the menufield_s control to memory.

Covering All the Bases

Because your new menufield_s control works in somewhat of an unorthodox manner (it commits data when the second mouse button or Esc key is pressed, instead of every time the control changes), you need to make sure you cover all your bases. In other words, there is one remaining way that a user could slip out of the menu without pressing the activating the K_MOUSE2 or K_ESCAPE variables, and that is by clicking the Back button directly. Presently, if a user were to type a new value into the Gender textbox and then click the Back button (which would signal K_MOUSE1, not K_MOUSE2), the changes made in the sex control would disappear.

To solve that problem, let's take a quick trip back to UI_Tweaks_MenuEvent, the function that handles the events for the remaining controls in your menu. Jump down to line 67, where the ID_BACK case is held, and make the following changes:

```
    case ID_BACK:
        TweaksSettings_SaveChanges(); // make sure that text control
is updated!
        UI_PopMenu();
        break;
```

The problem is solved with a simple call to TweaksSettings_SaveChanges, just before the menu disappears from view. Now all you need to worry about is making sure the control is added to the Tweaks menu, and pre-populating the textbox with the current value held in the sex Cvar. To accomplish both tasks, scroll down to line 139 near the end of UI_Tweaks_MenuInit, where all the controls are added to the menu, and make the following additions:

```
Menu_AddItem( &s_tweaks.menu, &s_tweaks.banner );
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.thirdPerson );
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.sex ); // new menufield_s!
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.framel );
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.framer );
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.back );

    // safe string-copy "sex" cvar into menufield_s control
    Q_strncpyz( s_tweaks.sex.field.buffer,
UI_Cvar_VariableString("sex"), sizeof(s_tweaks.sex.field.buffer) );
```

As you can see, right after the thirdPerson control is added, an additional call to Menu_AddItem allows the sex control to be added. Then, after all the controls are added to the s_tweaks.menu variable, a call to Q_strncpyz is made. Q_strncpyz is a custom function that performs a safe string copy from one variable to another, ensuring that there is a trailing zero at the end of the char array. The variable that will hold the copied string is the first parameter, s_tweaks.sex.field.buffer, which is a member in the menufield_s control. The value to be copied is obtained by making a call to UI_Cvar_VariableString, a system-call function that returns a console variable in the form of a string; the requested Cvar in this call is sex.

With all the changes in place, you should be able to build your new uix86.dll and give it a try. Once you have *Q3* loaded up, enter the Tweaks menu again and notice that the new control that reads "Gender." This is a free-form text box, so you can type whatever you want into it. Figure 9.9 shows me getting a little silly with the new control.

The menufield_s is not all that difficult to implement, as you have seen here. It offers a wide range of flexibility because it can also be constrained to allow only numbers to be entered, through the use of the QMF_NUMBERSONLY flag. As well, the menufield_s control supports the

Figure 9.9 *Entering a new gender in the Tweaks menu*

standard Copy and Paste shortcut keys; try copying some text into memory with Ctrl+C and then selecting your new text control and pressing Ctrl+V. The text should paste in, even if it came from a program outside *Q3*, such as Notepad.

The menuslider_s Control: Great for Parties

Without a doubt, the *menuslider_s* control defines coolness. It works just like a volume control on a stereo: it has a knob, often referred to as a *thumb,* that slides along a bar. Typically, the bar is narrow on one end and wide at the other, indicating that a value grows as the slider is moved from left to right. If you master the dark art of the menuslider_s control, you're sure to win friends and influence people. Let's add one to the Tweaks menu, modifying yet another Cvar, called `cg_fov`. This is a wacky Cvar that lets you control your player's field of view.

By default, the player's field of view, or *FOV,* is 90 degrees. Because the 90-degree range of view in a 3D world has to be cast onto a 2D surface

(your monitor), certain alterations are made so that the view fits appropriately. If the FOV were to increase dramatically, say to 130 degrees, the player would have a much larger range of view. However, because the 2D surface dimensions of the monitor are still the same, the alterations that are made to the final view are much greater, causing a stretched look. Similarly, reducing the FOV to below 90 degrees achieves a zoom effect, where the player sees less of the original view, but at a much closer distance (because the smaller view is stretched to fit on the same 2D monitor space). You can have a lot of fun with changing the FOV, so let's add a control to the Tweaks menu to do just that.

The guts of the menuslider_s control are simple and non-threatening, so much so that I'm going to place the definition here, right before your eyes:

```
typedef struct
{
   menucommon_s generic;

   float minvalue;
   float maxvalue;
   float curvalue;

   float range;
} menuslider_s;
```

Hopefully, that doesn't scare you too much. It starts with a menucommon_s variable, `generic` (as all good controls do), and then contains a series of floats. The first is `minvalue`, which holds the minimum value that the slider represents when the thumb is all the way to the left. The next float, `maxvalue`, represents the maximum value represented by the control, when the thumb is all the way to the right. As you might guess, `maxvalue` must be greater than `minvalue`. If you guessed that `curvalue` represents the current value of the slider, wherever the thumb is pointing, you've earned yourself another 50 bonus points.

The final float, `range`, is used to describe the increment that the slider uses to get from `minvalue` to `maxvalue`. So, for example, if you have a `range` of `1` assigned to your menuslider_s, the slider will only be capable of being set to whole numbers between your `minvalue` and `maxvalue`, like `1`, `10`, `15`, `50`, and so on (if your `maxvalue` was greater than

50). However, a range of 0.5 would allow the thumb to be set to values of 1.0, 1.5, 10.5, 20.0, 30.5, and so on.

Dropping the menuslider_s in

By now you should be familiar with stepping through the motions. Add a new ID_FOV variable definition at the top of ui_tweaks.c, under the previously defined variables.

```
#define ID_BACK             10
#define ID_THIRDPERSON      11
#define ID_SEX              12
#define ID_FOV              13
```

Next, add the control to the tweaks_t struct, calling it fov, and declaring it of type menuslider_s. An excerpt from tweaks_t should read like this:

```
    menuradiobutton_s     thirdPerson;
    menufield_s           sex;
    menuslider_s          fov;
```

The next change is an addition to the switch block in the UI_Tweaks_MenuEvent handling function. Right after the case for the ID_THIRDPERSON value, add a similar block to handle the new ID_FOV value.

```
    case ID_THIRDPERSON:
        trap_Cvar_SetValue( "cg_thirdPerson", s_tweaks.thirdPerson.cur-
value );
        break;

    case ID_FOV:
        trap_Cvar_SetValue( "cg_fov", s_tweaks.fov.curvalue );
        break;
```

There isn't anything secret happening here. Just as with the cg_thirdPerson Cvar, the cg_fov Cvar is set to the value currently held in s_tweaks.fov.curvalue, which will be the region to which the slider control currently points. Because I happen to be talking about members of the menuslider_s control, take a quick peek at Table 9.8, which lists the variable assignments necessary to use a menuslider_s control.

Table 9.8 Required Inits for menuslider_s

Variable	Value
`generic.type`	This member is assigned a value of `MTYPE_SLIDER`.
`generic.x`	This member sets the control's x location on the screen.
`generic.y`	This member sets the control's y location on the screen.
`generic.name`	This member holds the control's label, a text string drawn to the left of the control that does not change its x, y position.
`minvalue`	This member holds the slider's minimum value, which must be less than `maxvalue`.
`maxvalue`	This member holds the slider's maximum value.
`curvalue`	This member holds the slider's current value, as indicated by the thumb arrow on the slider.

With Table 9.8 as a guide, hop down to line 118 in ui_tweaks.c, where all the Tweaks menu's controls are initialized, and add in the initialization for the `fov` control:

```
    s_tweaks.fov.generic.type                = MTYPE_SLIDER;
    s_tweaks.fov.generic.name                = "Field of View:";
    s_tweaks.fov.generic.flags               = QMF_PULSEIFFOCUS |
                                               QMF_SMALLFONT;
    s_tweaks.fov.generic.callback            = UI_Tweaks_MenuEvent;
    s_tweaks.fov.generic.id                  = ID_FOV;
    s_tweaks.fov.generic.x                   = 320;
    s_tweaks.fov.generic.y                   = 170;
    s_tweaks.fov.minvalue                    = 1;
    s_tweaks.fov.maxvalue                     = 160;
    s_tweaks.fov.curvalue                    =
trap_Cvar_VariableValue( "cg_fov" );
```

You should have no problem identifying the values being assigned to the `fov` control here. The text that will describe the control reads "Field of View:" and, as you can see, the scope of the slider is from 1

to 160 (held in `minvalue` and `maxvalue`). This is because any value lower than `1` or higher than `160` assigned to `cg_fov` is automatically rounded to those numbers, respectively. The current value of the slider is set by reading in the current value of the `cg_fov` Cvar with a call to `trap_Cvar_VariableValue`.

The last addition is physically adding the control to the menu, and you do that on line 159, right after the `thirdPerson` and `sex` controls are added. Use the following snippet as a guide:

```
Menu_AddItem( &s_tweaks.menu, &s_tweaks.thirdPerson );
Menu_AddItem( &s_tweaks.menu, &s_tweaks.sex ); // new menufield_s!
Menu_AddItem( &s_tweaks.menu, &s_tweaks.fov ); // new menuslider_s!
```

And with that, you are done. Save your work, compile your uix86.dll, throw it in your MyMod folder, and fire up *Q3*. After entering the Tweaks menu, you should see your new Field of View control, as shown in Figure 9.10.

By default, the `cg_fov` value is 90, so the thumb should be somewhere near the middle of the slider. Try sliding it all the way to the right and

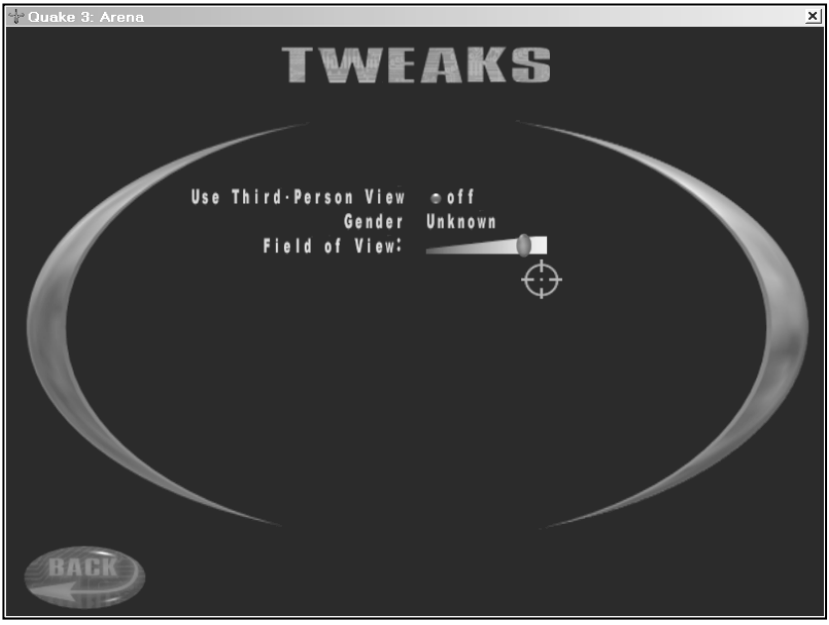

Figure 9.10 *Modifying the player's field of view in the Tweaks menu*

then starting up a game of *Q3*. I'll admit, it's a bit disorienting. I used to play the original *Quake* with a FOV setting of 130, so you can imagine what craziness I saw during a standard deathmatch.

NOTE

Allowing the player's field of view to be adjustable has come under fire recently—some players (who don't adjust their FOV) feel that doing so is cheating, because a wider FOV reveals more of the world. When *Q3* was developed, the `cg_fov` Cvar was not considered a "cheating" console variable; an option was added, however, for servers to prevent clients from adjusting their FOV during play.

Ultimate Power: menulist_s

The final control you'll be looking at in this chapter is the *menulist_s* control. The menulist_s control divvies out power to the user in the form of a list of elements that can be selected. This is the perfect control for a menu to allow someone to cycle through a specific set of items, in the event that he is unaware of all the values ahead of time. This saves the user from having to look up a value, or numerical representation of a variable, when he wants to make an adjustment in *Q3*'s user interface. For this section, you will add a menulist_s control that allows a user to select what type of shadow details he wishes to see.

In *Q3*, there are four settings for shadow details hidden away from the user in the Cvar `cg_shadows`. `0` denotes a value of off; when `cg_shadows` equals `0`, there are simply no shadows rendered by the engine. If `cg_shadows` is set to `1`, every moving or animated object that isn't a part of the level structure gains a soft shadow. If you look beneath the player's feet, you should see a inconspicuous, circular blur. The exact same shadow is applied to all objects; it is simply resized based on the object it is shadowing. When `cg_shadows` is set to `2`, however, things get pretty neat. Suddenly, all the shadows are dynamically built, based on the shape of the object that is being shadowed. So, the shadow of a player actually looks like the player, and moves as the player animates. Shadows of weapons and powerups also reflect the shape of their owners. Finally, if the `cg_shadows` Cvar is set to `3`, the complex shadows become darker, and faster to render.

NOTE

`cg_shadows 2` is rendered by the *Q3* engine using something called a *stencil buffer*. In a nutshell, a stencil buffer allows pixels to be drawn to the screen based on a user-defined value that references another set of pixels. Because the model of a player already exists in the 3D world, another version of that model (a squashed-flat 2D surface, for example) can be drawn using a stencil buffer, referencing the original model for shape, size, and dimensions. Using a stencil buffer is a fairly intensive task, and unless you have the latest and greatest hardware, you may experience some performance loss after turning `cg_shadows` to `2` in this tutorial.

Let's start by getting the gist of the menulist_s control. This control typically comes in two flavors, the Spin version and the List version. The Spin version works by drawing only one element of the list at a time; as the user clicks on it, the list cycles or spins from one element to the next. This version is perfect for tight-fitting quarters, where you need to conserve room in the layout of your menu. The other style, List, allows you to draw many elements in the list at once. The List version also allows you to use the control like a grid, containing not only multiple rows for each element in your list, but multiple columns for each row, giving the control a two-dimensional feel. Figure 9.11 shows one of the best uses of the List version of a menulist_s control, the server browser from within *Q3*.

Cold-Working the Spin Control

For this tutorial, you will use the simple and easy-to-implement Spin version of the menulist_s control. Before you start dropping code into your ui_tweaks.c file, however, let's take a moment to get a feel for the control.

The menulist_s struct is declared on line 187 of ui_local.h, and if you head over there, you should see something like the following snippet of code:

```
typedef struct
{
```

Figure 9.11 *The server browser using a menulist_s control*

```
    menucommon_s generic;

    int     oldvalue;
    int     curvalue;
    int     numitems;
    int     top;

    const char **itemnames;

    int     width;
    int     height;
    int     columns;
    int     seperation;
} menulist_s;
```

It should be no surprise that `generic` is the first member of the struct. (If it is a surprise, I get to take 50 of your bonus points away.) As you can see, there are a good number of integer declarations, such as `oldvalue`, `curvalue`, `numitems`, `top`, and as well, `width`, `height`, `columns`, and `seperation`. There is also a `const char` pointer, which itself is a pointer to `itemnames`. A pointer-to-a-pointer sounds complicated, but it really

isn't if you already understand what a pointer is. It is simply a variable that points to another variable that's doing some pointing of its own. Because standard C-style strings are typically held in a pointer-to-a-char, or *char**, it makes sense, that if you have a list of strings, and you want to be able to reference any particular string at one time, you will want to have a pointer to char*, which equates to a *char***.

For the Spin version of the menulist_s control, you'll need to initialize a certain set of members in the struct. Table 9.9 lists them.

TIP

The *const* keyword stands for "constant," meaning the value will be unchangeable. It is always good practice to write functions, structs, and so forth so that if they hold C-style strings that cannot be changed, they are declared as const. Many string-manipulation functions require a const_char* for exactly this reason, such as `strcpy` (copy one string to another) and `strcat` (add one string to the end of another string).

Table 9.9 Required Inits for menulist_s

Variable	Value
`generic.type`	This member is set to `MTYPE_SPINCONTROL`.
`generic.x`	This member sets the control's x location on the screen.
`generic.y`	This member sets the control's y location on the screen.
`generic.name`	This member holds the text label that is drawn to the left of the control.
`itemnames`	This member holds the list of elements to be cycled through by the control.
`curvalue`	This member references the currently selected element in the list, which maps to an index in the array held by `itemnames`.
`numitems`	This member holds the total number of elements in the list. For the `MTYPE_SPINCONTROL` style of menulist_s, this member does not need to be initialized or set; it is all handled automatically.

No surprises here, eh? Well, perhaps one: that funky pointer-to-a-pointer called itemnames. You need to provide a list of elements to the menulist_s control to allow the user to cycle through the list. This will be the first bit of code you lay into ui_tweaks.c for this final tutorial.

Scroll up to line 15, where the ID defines end, add a new one for the shadow control, and append the following code to it:

```
#define ID_FOV                  13
#define ID_SHADOW               14

static const char *shadow_types[] = {
    "No Shadows",
    "Standard",
    "Complex",
    "Dark Complex",
    0
};
```

After you have a new ID_SHADOW variable defined as 14, create the list for the menulist_s control by declaring a static const char* array called shadow_types. In the declaration of shadow_types, assign the values that will be used in the list. They are "No Shadows", "Standard", "Complex", and "Dark Complex", which will give you elements 0–3. (Remember, arrays in C start at 0, not 1!) Make a mental note that these elements map directly to the four values of cg_shadows that I explained earlier. The last element in the list is a 0, and indicates to the ui code that this list is now complete.

CAUTION

You must always specify 0 as your final element in a list that is supplied to a menulist_s control. If you don't . . . beware!

Don't forget to add your new control to the tweaks_t struct declaration, like so:

```
    menufield_s         sex;
    menuslider_s        fov;
    menulist_s          shadowDetail;
    menubitmap_s        back;
} tweaks_t;
```

For this tutorial, I call the menulist_s control shadowDetail. Make sure you set aside a way for the Tweaks menu event handler to deal with

someone clicking on the menulist_s control. Do that in UI_Tweaks_MenuEvent, on line 83, right after ID_FOV handler:

```
        trap_Cvar_SetValue( "cg_fov", s_tweaks.fov.curvalue );
        break;


    case ID_SHADOW:
        trap_Cvar_SetValue( "cg_shadows", s_tweaks.shadowDetail.curvalue );
        trap_Cvar_SetValue( "r_stencilbits", 8 );
        trap_Cmd_ExecuteText( EXEC_APPEND, "vid_restart;" );
        break;
```

Here, you actually execute a couple of functions if the ID_SHADOW ID passes into the event handler. First, the cg_shadows Cvar is set to the current value of the menulist_s control. Remember, curvalue references the index of the array, not the text string in that index. So if the menulist_s control currently reads "No Shadows," then curvalue will actually equal 0 (and 0 is what will be passed back to the cg_shadows Cvar). Then, another Cvar called r_stencilbits is set to a value of 8. This is a requirement of the complex shadow type (when cg_shadows equals 2), so just to be quick and dirty, you can go ahead and set it to 8 in each case. Finally, a system-call function named trap_Cmd_ExecuteText is called, passing in EXEC_APPEND as the first parameter and vid_restart as the second. vid_restart is actually a console command that forces *Q3* to re-initialize the 3D rendering engine from scratch, which is required when shadow types and stencil-buffer depths change.

The menulist_s control will need to be initialized; you know what you have to do. Hop down to line 145 in ui_tweaks.c, and add the following code after the fov control ends to get your shadowDetail control freaky-fresh and fly:

```
    s_tweaks.fov.maxvalue                   = 160;
    s_tweaks.fov.curvalue                   = trap_Cvar_VariableValue
( "cg_fov" );


    s_tweaks.shadowDetail.generic.type      = MTYPE_SPINCONTROL;
    s_tweaks.shadowDetail.generic.name      = "Shadow Detail:";
    s_tweaks.shadowDetail.generic.flags     = QMF_PULSEIFFOCUS |
QMF_SMALLFONT;
    s_tweaks.shadowDetail.generic.callback  = UI_Tweaks_MenuEvent;
```

```
    s_tweaks.shadowDetail.generic.id          = ID_SHADOW;
    s_tweaks.shadowDetail.generic.x           = 320;
    s_tweaks.shadowDetail.generic.y           = 190;
    s_tweaks.shadowDetail.itemnames           = shadow_types;
    s_tweaks.shadowDetail.curvalue            = trap_Cvar_VariableValue(
"cg_shadows" );
```

Let's go over the nitty gritty: `MTYPE_SPINCONTROL` is your `generic.type`, while the label of the control will read "Shadow Detail:" (set in `generic.name`). The `generic.flags`, `generic.callback`, and `generic.id` should be clear, as well as the `generic.x` and `generic.y` values. As expected, the `itemnames` member is assigned to the static const char* array you created, holding each value in the list of shadow types. Finally, the currently selected element in the list is assigned to `curvalue`, and is polled by returning the value of `trap_Cvar_VariableValue`, looking at the Cvar `cg_shadows`.

Don't forget to cross your t's and dot your i's; the menulist_s control will do no good to you if you don't add it to the context of the `s_tweaks.menu` variable. Line 186 will be your final code adjustment:

```
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.fov ); // new menuslider_s!
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.shadowDetail ); // new
menulist_s!
    Menu_AddItem( &s_tweaks.menu, &s_tweaks.framel );
```

Congratulations, you are now the owner of a brand-new baby . . . er . . . menulist_s control. Fire up the compiler, build a new uix86.dll, copy it to MyMod, and load *Q3*. If all goes well, entering the Tweaks menu should reveal a new Shadow Detail: control (see Figure 9.12).

If you click on the menulist_s control, *Q3* will blink and grind for a moment as it restarts the video renderer (remember the call to `vid_restart`?). Then the *Q3* menu should be visible once again, and if you return to "Tweaks" a second time you should see the next value in the list. Try setting the value to "Complex" and then firing up a level. Take a look at the shadows underneath the weapons and powerups; you'll see that they match the shape and size of the objects. As mentioned earlier, detailed shadows required specific stencil-buffering capabilities that some video cards lack, so if you don't see the shadows with Complex or Complex Dark, you can always go back to Standard, or turn shadows off completely with No Shadows.

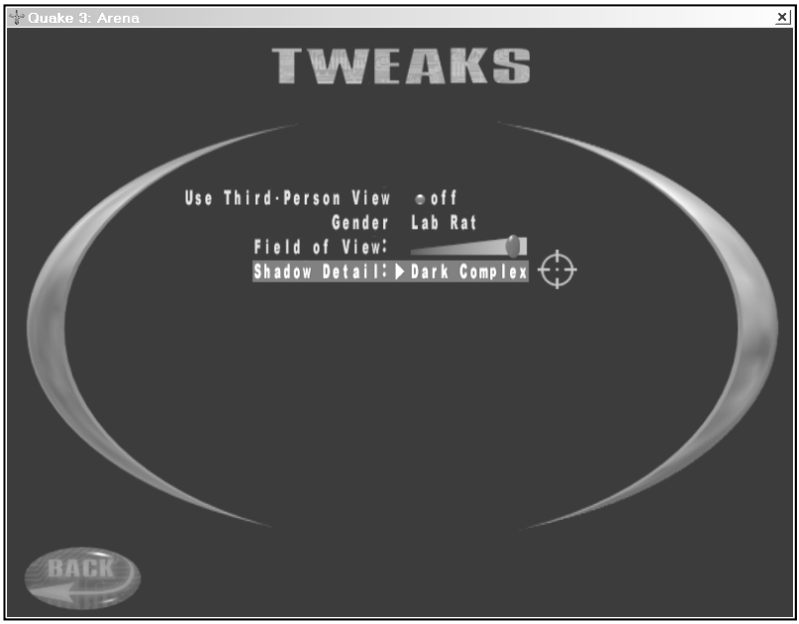

Figure 9.12 *The new Shadow Detail control in the Tweaks menu*

Summary

This was one epic chapter! You now have the ability and tools to continue in your exploration of the `ui` code. You should understand that all menus begin with a menu framework, and each menu contains a set of controls that allow the user to interact with the interface. Each control is unique and specifically tasked for different uses—menulist_s allows users to cycle through multiple elements, while menuradiobutton_s allows users to turn a value on or off. I encourage you to revisit the `ui` code and try working with more control settings, layouts, and investigating how the existing *Q3* menu system is implemented. There is no better way to learn than by seeing how those before you have created.