CHAPTER 10

Enhancing DTF

By now you've successfully implemented a new game type into *Q3* using new rules, new item behaviors, and new scoring systems. That may not, however, be enough to satisfy players. For example, in your new *DTF* game type, how can a player tell which sigils are currently held by the red team as opposed to the blue team? You've also introduced an element of randomness to your mod by dynamically creating a third sigil spawn point, and many players already familiar with *CTF* maps are going to wonder, "Where the heck is the third flag?" In this chapter, I'm going to show you how you can polish your mod, offer ways to solve the problems possed here, and assist you with the setup of *DTF* from within the *Q3* user interface.

Adding Sigil Status to the HUD

When I first started discussing the *DTF* game type, I used *Unreal Tournament*'s *Domination* as a basis of reference. I also included an image which showed how the game displayed the status of the three control points. Down the left side of the player's HUD, three icons are rendered that indicate, by their shape and color, both the control point and who controls it. You can accomplish a similar layout using the three icons to represent standard *CTF* flags in their "at home" state simply by changing the color of each icon based on which team holds each sigil.

Filling in the Missing Game Code

The first task you need to complete is filling in the missing code that you began to lay out in Chapter 7. A few functions were left empty and a few variables went undeclared because they weren't needed at the time; they dealt specifically with communicating information from the game module to the `cgame` module. Now that you will need to tell `cgame` what the status of the three sigils are, this code must be in place.

Start by heading back to g_team.c and jumping to line 185. You should see two variable definitions, like so:

```
static char ctfFlagStatusRemap[] = { '0', '1', '*', '*', '2' };
static char oneFlagStatusRemap[] = { '0', '1', '2', '3', '4' };
```

These two character arrays contain a single char for each value, mapping to a status that a flag could have referenced by the flagStatus_t enum. For example, in standard *CTF*, flags can be `FLAG_ATBASE`, `FLAG_TAKEN`, or `FLAG_DROPPED`, the values of those variables being `0`, `1`, and `4`, respectively. Because a flag's status is wrapped up in a config string (which I will get to later), it will be communicated in the form of a single character. The character is simply a char of the numerical equivalent; that is, 1 is `'1'`, 2 is `'2'`, and so on. One char will exist for every appropriate index in flagStatus_t. Although this sounds like an unorthodox way of handling values, the methodology will soon be made clear.

You use sigilStatus_t to represent the three statuses of a sigil in *DTF*: `SIGIL_ISWHITE`, `SIGIL_ISRED`, or `SIGIL_ISBLUE`. Because these variables actually hold the values `0`, `1`, and `2`, you'll need to create a char array that includes the ASCII representations of these three values. Above the declaration of these two char arrays, create a new one, like so:

```
static char dtfSigilStatusRemap[] = { '0', '1', '2' }; // maps to
sigilStatus_t
```

Now that you have a char array that will remap your enum values into actual characters, let's put them to use in the function we left behind in Chapter 7: `Team_SetSigilStatus`.

After all the hacking, inserting, modifying, and such that g_team.c has gone through, functions will more than likely have been moved around in this file. If you've followed along in the book from previous chapters, `Team_SetSigilStatus` should be located near line 287 now. This function was left empty because it handles the communication of the sigil statuses to the `cgame` code, something you didn't need in Chapter 7. You do now, though! Go ahead and fill in the missing code so that the function reads like this:

```
void Team_SetSigilStatus( int sigilNum, sigilStatus_t status ) {
        qboolean modified = qfalse;

        // update only the sigil modified
        if( teamgame.sigil[sigilNum].status != status];
            teamgame.sigil[1].status = status;
              modified = qtrue;
      }

      if( modified ) {
            char st[4];

            // send all 3 sigils' status to the configstring
            st[0] = dtfSigilStatusRemap[teamgame.sigil[0].status];
            st[1] = dtfSigilStatusRemap[teamgame.sigil[1].status];
            st[2] = dtfSigilStatusRemap[teamgame.sigil[2].status];
            st[3] = 0;

            trap_SetConfigstring( CS_SIGILSTATUS, st );
      }
}
```

This function borrows functionality from its sister function, `Team_SetFlagStatus`, which is used by standard *Q3*. In this function, the sigil that is having its status set is referenced by `sigilNum`, an integer passed into `Team_SetSigilStatus` that represents the index of the `teamgame.sigil` array. This means that the only valid values can be `0`, `1`, and `2`. The `status` variable is also passed in, which represents one of the three sigilStatus_t values. If the specific sigil's status does not match the status passed in, the new status is assigned to the sigil, and a flag that the sigil was modified is set to `true`.

In the second half of the function, the `modified` flag is examined to see whether a sigil's status was changed. If so, a new char array called `st`, with a length of `4`, is created. Then, for the first three indexes of `st` (`0` through `2`), the status of the sigil found in the same index is used as the index of your new `dtfSigilStatusRemap` variable, which is assigned back to `st`. Ugh! Does that sound complicated? Let me set it down in layman's terms:

- Each element in `st` is a char, like `'1'` (not the value 1).
- Each element in `dtfSigilStatusRemap` is a char, like `'1'`.
- Each element in `dtfSigilStatusRemap` is accessed by a numerical index, such as `0`, `1`, or `2`.
- Each status of the `teamgame.sigil` array is a value, represented by the enum sigilStatus_t (which can either be `0`, `1`, or `2`).

- Therefore, the value of any status in the `teamgame.sigil` array can be used as an index to `dtfSigilRemap`, producing a char, such as `'1'`, which can be assigned back to `st`.

Phew! Hopefully those notes clarify what you are doing—essentially, you're assigning characters, which are ASCII representations of values, to a variable called `st`. The last element of the `st` array is set to `0` to indicate that the array has ended.

The final line of this function makes a system-function call:

```
trap_SetConfigstring( CS_SIGILSTATUS, st );
```

This passes the `st` variable from the `game` code to the `cgame` code, with the flag of `CS_SIGILSTATUS`, indicating that the variable's values will be used for determining the status of sigils. So now the question remains: Where did `CS_SIGILSTATUS` come from, and what the heck is a config string?

Making the Config String Work for You

In *Q3*, the `game` and `cgame` code must keep communication going between them. One of the techniques used to allow values to be passed back and forth between the modules is the implementation of the *config string*. Basically, a config string consists of an array of characters that are ASCII representations of numerical values. This section will show you that by using a clever bit of math, the characters in a config string can be converted to numbers quickly and efficiently, without any extra function calls.

In order to set up a config string, you will first need to declare your new config-string identifier. Open bg_public.h and scroll to line 65, where you should see a number of config strings already being declared. After `CS_ITEMS` is declared, go ahead and add a new line, and add your missing `CS_SIGILSTATUS` variable:

```
#define CS_SIGILSTATUS              29          // dtf configstring
```

That wasn't so hard! Now, your `Team_SetSigilStatus` function will be able to appropriately identify the `st` variable being passed to cgame, because the entire code base now knows what `CS_SIGILSTATUS` refers to.

Before you jump over to `cgame`, can you think of one other place that you will want to use `CS_SIGILSTATUS`? If you guessed `Sigil_Touch`, you are correct, sir! Sigil_Touch has the responsibility of updating the sigils on the server side of things when players touch them. Naturally, it's also going to need to communicate those statuses back to the client-side portion of the code.

Jump back to g_team.c and scroll to line 1046, which is where you should find Sigil_Touch. Start by declaring a new variable, an integer named sigilNum, at the start of the function:

```
int Sigil_Touch( gentity_t *ent, gentity_t *other ) {
    gclient_t *cl = other->client;
    int sigilNum = 0;
```

sigilNum is going to be used to figure out which sigil was passed to Sigil_Touch, via the ent parameter. Next, after the overflow protect on ent->count is performed, add this snippet:

```
    // find the index of the sigil referred by ent
    while ( sigilNum < MAX_SIGILS && teamgame.sigil[sigilNum].
entity != ent )
        sigilNum++;
```

Because Team_SetSigilStatus requires an integer (the proper index for teamgame.sigil), and the only reference to a sigil is via the ent variable, the proper index is retrieved by accruing an integer representing the index in the teamgame.sigil array and comparing it with the passed-in ent variable.

The last change you need to make to Sigil_Touch is within each of the two changes that sigil undergoes when being converted to either red or blue. Make this change by making a call to Team_SetSigilStatus as the first line of code within both sigil conversion blocks. The code for converting a sigil to red should read as follows:

```
    if ( cl->sess.sessionTeam == TEAM_RED && ent->s.powerups
!= PW_SIGILRED )
    {
        Team_SetSigilStatus(sigilNum, SIGIL_ISRED);
```

whereas that for converting a sigil to blue should read like this:

```
else if ( cl->sess.sessionTeam == TEAM_BLUE && ent->s.powerups
!= PW_SIGILBLUE )
    {
        Team_SetSigilStatus(sigilNum, SIGIL_ISBLUE);
```

Note that the previously incremented integer `sigilNum` is used in both cases, indicating the proper index of the `sigil` array, within the `teamgame` variable.

Nice work. You now have all the `game` code in place to handle communicating the status of the three sigils back to the `cgame` code, via the use of a config string. Actually parsing out the values of the config string on the `cgame` side will be a different story altogether. . . .

Prepping cgame for the HUD Update

In order to allow `cgame` to understand what data is coming down from `game`, a few preparations are in order. This new config string will be used to update the scoreboard that you will draw for players of *DTF*, so you will have to go through the motions necessary to set that information up ahead of time. The new scoreboard will need icons that represent the status of the sigils; in addition, `cgame` will need to know what sigils are. Take a moment now and get all your ducks in a row.

Your first stop is cg_local.h. You'll need to create a handle to the icon that will represent a sigil in its initial white state. The red and blue states already exist; you'll reuse the shaders that are created for the *CTF* flags. Scroll down to line 652, and after the `redFlagShader` and `blueFlagShader` arrays are declared, add one called `sigilShader`:

```
    qhandle_t    redFlagShader[3];
    qhandle_t    blueFlagShader[3];
    qhandle_t    flagShader[4];
    qhandle_t    sigilShader; // dtf icon of sigil in white state
```

You may be wondering why the `redFlagShader` and `blueFlagShader` variables are declared as arrays. This is done because in *CTF*, the icons that represent flags come in multiple flavors. There is a standard flag icon, one with an X through it (to indicate the flag was stolen) and one with a ? through it (to indicate that it was dropped somewhere in the map). You do not need multiple icons for your shader; the standard white flag icon will do nicely.

The next step involves getting your new handle pre-cached when a `GT_DTF` game type is initializing in *Q3*. You may recall that the precaching functions are handled in cg_main.c. Open that file and scroll down to about line 883, where various *CTF*-related shaders are initialized, as long as the game type is `GT_CTF`. You will want use the same initializations for your HUD, because they include the initialization of the `redFlagShader` and `blueFlagShader` variables I was just discussing. Find the line of code that reads

```
if ( cgs.gametype == GT_CTF || cg_buildScript.integer ) {
```

and change it to the following:

```
if ( cgs.gametype == GT_CTF || cgs.gametype == GT_DTF ||
cg_buildScript.integer ) {
```

This will allow the shaders to be initialized in both `GT_CTF` and `GT_DTF`. You aren't quite done yet, however—the `sigilShader` handle is still not initialized. Jump down a few lines to 892, where the `blueFlagShader` is set up, and add your `sigilShader` definition like so:

```
        cgs.media.blueFlagShader[2] = trap_R_RegisterShaderNoMip(
"icons/iconf_blu3" );
        cgs.media.sigilShader = trap_R_RegisterShaderNoMip(
"icons/iconf_neutral1" );
```

Excellent; the white flag icon is now in place, representing your initial sigil status as it first appears in the game. Now that you have memory set aside for all three icons that will represent the status of the game's sigils within *DTF*, it is time to inform `cgame` what a sigil really is. To do this, return to cg_local.h and scroll to line 1008, which should put you right in the heart of the cgs_t struct declaration. Recall that cgs_t is the data type of `cgs`, the global client-side game variable that holds everything from the current game type and frag limit to the pre-cached shaders used in the HUD, as well as a direct connection to the state of the game as it exists on the server, for synchronization.

Around line 1008, you should see declarations of integers that will represent the status of the red and blue flags, as well as the white flag (in the Mission Pack). Go ahead and add a new line, and declare an integer for the status of your sigils. Because there are three sigils in a game of *DTF*, you have full clearance from me to declare the variable as an array:

```
    int      redflag, blueflag;   // flag status from configstrings
    int      flagStatus;
    int      sigil[MAX_SIGILS];   // dtf sigil status from configstrings
```

Once the `sigil` array is declared within the cgs_t struct, you are free to access it via `cgs.sigil`. Go ahead and do that now by properly setting them to –1 when a new game first starts up. The function that first sets up the cgame code is `CG_Init`, found in cg_main.c at around line 1840. Dive into that function now and scroll farther down, near line 1868, where the red, blue, and white flags are already being initialized. Add a line of code to set the three sigils in the same fashion, like so:

```
    cgs.redflag = cgs.blueflag = -1; // For compatibily, default to
unset for
    cgs.flagStatus = -1;
    cgs.sigil[0] = cgs.sigil[1] = cgs.sigil[2] = -1; // dtf sigils
reset
```

You might remember that you perform a similar action on the `game` side of things, within `Team_InitGame`. It's important to keep data as synchronous as possible between `game` and `cgame`.

Parsing Out Config Strings in cgame

Your structure for dealing with the sigils in the `cgame` code is now in place. The next task to perform is the actual parsing out of the values held in the config string once they come over from `game`. When I began to describe config strings earlier in this chapter, I alluded to a fancy technique used to extract the required data from them. Following is the secret of that technique.

One of the ways in which a programmer can use the various letters, numbers, and symbols that a computer supports is by referring to an organized table called *ASCII*. The ASCII table starts off with a set of funny symbols, including happy faces, musical notes, and pointing arrows, eventually leading into punctuation marks. Then, starting at the 49th element, the characters 0 through 9 come up, followed by the letters of the alphabet. Because the characters that represent the numbers 0 through 9 are sequential, and they are ordered in an array, the distance from a given number back to the 0 character will always equal that specific number, as long as the number is from 0 to 9.

For example, suppose you have a char variable holding the character 5. If you subtract the character 0 from it, you will get a numeric representation of the distance, which is 5! Because config strings are a conglomeration of characters that represent numbers, this simple and tricky technique is the fastest way to extract numerical data from them.

NOTE

This technique of converting chars into integers is fast (and cool) but will not work for any value above nine. Those values include two characters (such as 10), making them strings or *char arrays*. The solution to this problem is simply to extract the char array from the config string with a call to `CG_ConfigString`, and then, pass that new value to `atoi`, a standard C function that converts strings to integers.

Let's try this fly math out in the function `CG_SetConfigValues`, located in cg_servercmds.c. `CG_SetConfigValues` is responsible for initializing a good deal of similar client-side variables via the config string. Open cg_servercmds.c and scroll to line 188, where the config strings are parsed out for *CTF*:

```
if( cgs.gametype == GT_CTF ) {
    s = CG_ConfigString( CS_FLAGSTATUS );
    cgs.redflag = s[0] - '0';
    cgs.blueflag = s[1] - '0';
}
```

Here, the `cgs.gametype` variable is queried for a value of `GT_CTF`. If a game of *CTF* is detected, a variable `s` (which is a pointer of type const char) is assigned the value of `CG_ConfigString`, using `CS_FLAGSTATUS` as the ID of the requested config string. Then, the `cgs.redflag` and `cgs.blueflag` variables are assigned their status by looking at the matching index of the char array, as pulled from `CG_ConfigString`. Notice that in both cases, the char found in each index of the `s` char array has the character '0' subtracted from it, returning a numerical representation back to `cgs.redflag` and `cgs.blueflag`. This final number will represent the flag's status, either `0`, `1`, or `4` (remember the remap?).

Go ahead and make the following addition, directly after the code listed previously:

```
else if ( cgs.gametype == GT_DTF ) {
    s = CG_ConfigString( CS_SIGILSTATUS );
    cgs.sigil[0] = s[0] - '0';
```

```
        cgs.sigil[1] = s[1] - '0';
        cgs.sigil[2] = s[2] - '0';
    }
```

No surprises here; you perform exactly the same task for *DTF* as was performed for *CTF*. The only difference is that you use CS_SIGILSTATUS as your identifier of the requested config string, and there is a third cgs.sigil to assign. Other than that, you're practically stealing code at this point! Don't feel guilty; learning by example is a good way to see how other coders implement solutions.

In the same file, there's a second function called CG_ConfigStringModified, which is called by *Q3* when a server command is issued by the game code. It alerts cgame that config strings have changed. A good example of this is when one of the new *DTF* sigils is touched; you'll recall that in game, the function Team_SetSigilStatus creates an updated config string and passes it to cgame, via trap_SetConfigString. Hop down to line 330, in the midst of CG_ConfigStringModified, and add this code snippet after GT_CTF and CT_1FCTF are dealt with:

```
    else if ( num == CS_SIGILSTATUS ) {
        if ( cgs.gametype == GT_DTF ) {
            cgs.sigil[0] = str[0] - '0';
            cgs.sigil[1] = str[1] - '0';
            cgs.sigil[2] = str[2] - '0';
        }
    }
```

Here, num represents the ID of the config string being queried. If the ID is CS_SIGILSTATUS, then you can perform the necessary updates to the cgame version of the sigils. As before, str (which is now the reference to the value held in the config string) has the '0' character subtracted from each of its indexes, and is assigned to the appropriate cgs.sigil variables.

The Sigil Status HUD Comes to Life

At this point, you're successfully keeping cgame in sync with game, so that both modules know the status of the sigils. The last task is to physically create the scoreboard and render it to the player's HUD, showing the appropriate status of each sigil. Start by opening cg_draw.c and scrolling to line 527. This will put you in the heart of `CG_DrawStatusBar`, the function responsible for drawing the HUD. After the check performed on `cg_drawStatus.integer`, add a new function call, like so:

```
    if ( cg_drawStatus.integer == 0 ) {
        return;
    }

    // draw the dtf sigils
    if (cgs.gametype == GT_DTF)
        CG_DrawSigilHUD();
```

Here, a very simple check is made on `cgs.gametype`, and if the value is `GT_DTF`, then a call is made to `CG_DrawSigilHUD`, a function you will write next. Drawing icons to the HUD is a fairly straightforward task, done by using a function called `CG_DrawPic`, which is defined as follows:

```
void CG_DrawPic( float x, float y, float width, float height, qhandle_t
hShader )
```

The function `CG_DrawPic` uses an x and y location to position the image on the HUD, as well as a width and height to resize the image on-the-fly (if needed). It also takes a handle to a shader. This function automatically translates your positions into the appropriate 640 × 480 coordinate system, so it is extremely easy to use. Here is the body of `CG_DrawSigilHUD`, which you can place in cg_draw.c directly above `CG_DrawStatusBar`:

```
void CG_DrawSigilHUD( void ) {
    int i, x=10, y=120;

    for (i=0; i<MAX_SIGILS; i++) {

        switch (cgs.sigil[i])
        {
        case SIGIL_ISWHITE:
            CG_DrawPic( x, y, 24, 24, cgs.media.sigilShader );
            break;
        case SIGIL_ISRED:
            CG_DrawPic( x, y, 24, 24, cgs.media.redFlagShader[0] );
            break;
            case SIGIL_ISBLUE:
                CG_DrawPic( x, y, 24, 24, cgs.media.blueFlagShader[0] );
                break;
            }

            y+= 80;
        }
}
```

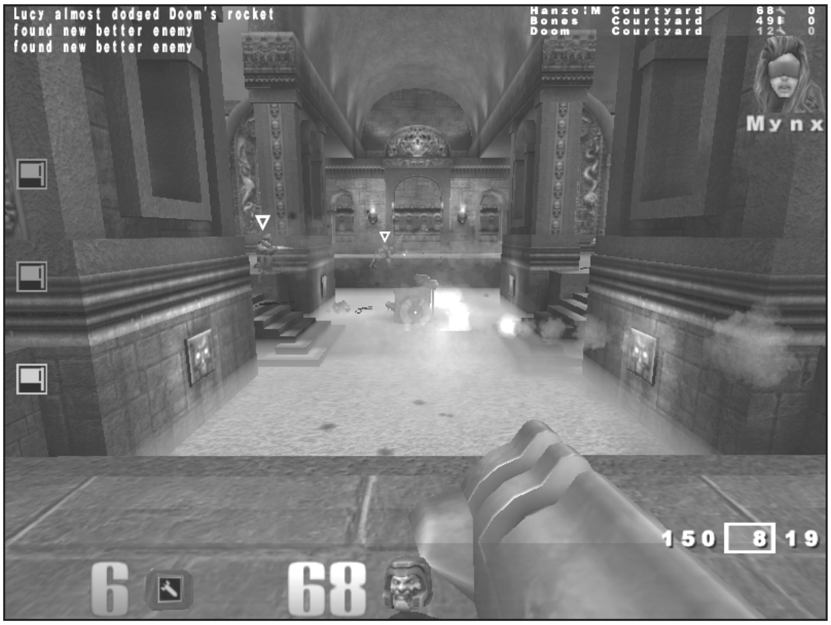
In this function, an initial coordinate is set to 10 × 120, which will position the drawing to the far left of the screen, a bit down from the top. Then, a loop is performed over the three sigils, switching off the status of each sigil in the loop. `CG_DrawPic` is called, passing in the current x and y values and resizing the image to 24 × 24 (because the original flag status icons are 32 × 32). Finally, the appropriate shader is passed based on the status of the sigil. `sigilShader` is used for `SIGIL_ISWHITE`, while `redFlagShader[0]` and `blueFlagShader[0]` are used for `SIGIL_ISRED` and `SIGIL_ISBLUE`, respectively. Remember that you're reusing the flag status icons from *CTF*, hence the reference to the `redFlagShader` and `blueFlagShader` arrays.

As the loop iterates, the value of the y location is incremented by 80, which pushes each icon 80 pixels down the side of the screen, creating all three icons in a vertical row. If you build your cgamex86.dll and qagamex86.dll files and launch *Q3* with them, turning on the `GT_DTF` game type, you should see the icons display in a vertical row, as shown in Figure 10.1

Very nice. Your work is starting to feel like a polished mod. Not only do you offer a new game type and a new way of dealing with flags, you also offer the player an updated HUD, which better communicates the status of the game. Players can now get a quick visual of the flags and determine how many are held by their team.

Adding a Flag Locator

With the status of each sigil in place on the HUD, players can get a quick update by glancing at the left side of the screen to see how many flags their team holds. Because you have introduced the possibility of a third flag into the mix, however, players will want to know



Lucy almost dodged Doom's rocket
found new better enemy
found new better enemy

```
Hanzo:M  Courtyard   68   0
Bones    Courtyard   49   0
Doom     Courtyard   12   0
```

Figure 10.1 *The new* DTF *HUD*

which one is which. Simply showing a red flag icon is not enough; is it the flag in the red base, the blue base, or in the middle of the map? The solution to this predicament is to build a flag locator that will act as a point of reference for the player in his quest to hold all three flags. The locator will draw tiny flag icons on the HUD, and will rotate around the player's view screen like a compass, always pointing to the flags. This will assist players in figuring out where they need to be.

Getting to Know Cvars

The flag locator that will be built in this tutorial is a new and exciting addition to the HUD. That said, not all players will want to use it. For this reason, you will use Cvars to make this update a modifiable selection for the user. I briefly mentioned Cvars way back in Chapter 5; you should recall that *Cvars* is short for *console variables*. A Cvar is a special set of variables that give the player a direct hook right into the code so that they can turn things on or off, or set specific options to new values. Cvars help make *Q3* more configurable, which in turn gives more power back to the player.

Cvars typically come in all flavors. Some hold numerical values, while others hold strings. There are fixed sets of Cvars that are hard-coded into the *Q3* EXE file, meaning that they will always be present in any game or mod. Then, there are module-specific Cvars that are defined in the game source: game Cvars are typically identified by a `g_`, `cgame` Cvars start with `cg_`, and UI Cvars have `ui_` at the beginning. Let's take a look at what makes up the data portion of a Cvar by examining its creation in `cgame` (because your flag locator will ultimately end up here).

Line 1064 of cg_local.h marks the declaration of all the client-side Cvars currently in *Q3*. Each one is declared with the `extern` keyword, meaning the variable will be accessible to other files in the code base. The data type of each Cvar is a *vmCvar_t*; the declaration of that type can be found line 934 of q_shared.h:

```
#define    MAX_CVAR_VALUE_STRING    256

typedef int    cvarHandle_t;

typedef struct {
    cvarHandle_t   handle;
    int             modificationCount;
    float           value;
    int             integer;
    char            string[MAX_CVAR_VALUE_STRING];
} vmCvar_t;
```

It's not a very complicated variable type, but it's important nonetheless. Typically, you will access the value of a particular Cvar by referencing either its `value`, `integer`, or `string` members, depending on the type of data it holds.

Once a Cvar is declared, it needs to be poured into the main list of Cvars that *Q3* keeps a record of during each game. You do this by adding the Cvar to an array called `cvarTable`, which itself is of data type *cvarTable_t*:

```
typedef struct {
    vmCvar_t    *vmCvar;
    char         *cvarName;
    char         *defaultString;
    int         cvarFlags;
} cvarTable_t;
```

The cvarTable_t struct is declared on line 180 of cg_main.c (for cgame Cvars). Within it, a pointer to a vmCvar_t exists, which will map to the declaration of the variable mentioned earlier. cvarName is a char pointer, which will hold the name of the Cvar as it would be read from or written to the console (such as cg_shadows). The next member, defaultString, is another char pointer that holds the default value of the Cvar when it is first fed into *Q3*. It could be anything, from a number, such as 1, to a complete word, such as a player's name (as in Casey|M). Finally, the cvarFlags variable is an integer that can hold a combination of different Cvar-related flags, which tells *Q3* how to handle the Cvar once it is a part of the game. Table 10.1 lists the available flags.

For the most part, any of the Cvars you create will use only one or two of the flags listed in Table 10.1 (three at the most!). To see an example of an entry in the cvarTable array, take a look at this snippet that initializes cg_fov, the Cvar you played with in Chapter 9:

```
{ &cg_fov, "cg_fov", "90", CVAR_ARCHIVE },
```

In this element of the cvarTable array, the cg_fov variable is assigned with a matching "cg_fov" string (which is how users will see the variable displayed in the console). The default value is 90, and the CVAR_ARCHIVE flag is used, meaning that *Q3* will write the current value of cg_fov to the player's config.cfg file, to be used when the game is launched at a later time.

TIP

Not to be confused with config strings, config.cfg is a physical file, written to the hard drive by *Q3*, that holds a series of Cvars and their appropriate values. This file is also parsed by *Q3* when a new game is launched, overriding the programmatic default values with those found in the file. Each game directory within /quake3/ should have its own config.cfg file, including your MyMod folder.

After all the necessary Cvars are added to the cvarTable array, the array is fed directly into *Q3*, registering each Cvar and making it official. This is done via the CG_RegisterCvars function in cgame. Cvar

Table 10.1 Cvar Flags

Flag	Value
`CVAR_ARCHIVE`	This flag causes the Cvar to be saved and written to a config file when *Q3* exits.
`CVAR_USERINFO`	This flag indicates that the Cvar should be communicated to the server when changed.
`CVAR_SERVERINFO`	This flag describes a Cvar that handles some server-related information, such as game type, frag limit, and so on.
`CVAR_SYSTEMINFO`	This flag describes a Cvar holding miscellaneous system-related information.
`CVAR_INIT`	This flag prevents the Cvar from being updated via the console. It can still be set by the command line, however.
`CVAR_LATCH`	This flag latches the Cvar to the actual game code, meaning it will not be accessible or modifiable unless the C code in the game properly initializes it.
`CVAR_ROM`	This flag stands for *read-only*; any Cvar having this flag will not be modifiable at all.
`CVAR_USER_CREATED`	This flag is assigned to Cvars manually by the player using the `set` command in the console.
`CVAR_TEMP`	This flag describes a Cvar that is temporary; it will not be saved by *Q3* when the game exits.
`CVAR_CHEAT`	This flag denotes the Cvar as a cheating variable; it will not be modifiable unless cheats are enabled on the server.
`CVAR_NORESTART`	This flag indicates that the Cvar will not be reset when `cvar_restart` is issued to the console.

initialization is not arbitrary; it goes through the exact same motions in `game` and `ui` as well, first being declared as a vmCvar_t variable, then placing that variable in an element of some global Cvar array, after which the entire array is registered by a specific function. That's Cvars in a nutshell.

Adding a Cvar for the Flag Locator

Now that you're a certified expert in the use of Cvars, you can start this flag-locator tutorial by creating a Cvar that the player will use to toggle the display on or off. Start by opening cg_local.h and scrolling to about line 1154, which puts you in the midst of similar Cvar declarations. Remember, this is the `extern` declaration of the variable, notifying the other files of its eventual existence. You'll need to make the formal declaration within the file that uses it. Start by adding its `extern` declaration after `cg_trueLightning`:

```
extern    vmCvar_t        cg_trueLightning;
extern    vmCvar_t        cg_sigilLocator; // dtf locator
```

This `extern` declaration tells the compiler to keep its eyes open for `cg_sigilLocator` being used in other files during compilation. The other declaration of the variable will take place on line 164 of cg_main.c, where the rest of the client-side Cvars are officially declared. Just as before, make your new Cvar's declaration right after `cg_trueLightning`:

```
vmCvar_t    cg_trueLightning;
vmCvar_t    cg_sigilLocator; // dtf locator
```

With this declaration in place within cg_main.c, you're now free to properly initialize and register your Cvar by placing it the `cvarTable` array, found near line 188. Go ahead and add it to the very end of the array (and don't forget to add a comma at the end of the previous element, because `cg_trueLightning` is no longer the last element in the array!).

```
    { &cg_trueLightning, "cg_trueLightning", "0.0", CVAR_ARCHIVE},
    { &cg_sigilLocator, "cg_sigilLocator", "1", CVAR_ARCHIVE} // dtf
sigil locator
```

This code tells *Q3* that `cg_sigilLocator` is the name of the new Cvar, it'll default to `1`, and its setting will be saved into the config.cfg file when *Q3* exits. That's it! Adding Cvars to *Q3* is an extremely easy process; I encourage you to add more where applicable. The more options you can give the user, the better the odds that your mod will appeal to a wide variety of people, because they will be able to tweak your mod to their liking.

Adding the Flag-Locator Functions

Two functions are involved in drawing the flag locator to the screen. The first of these functions is CG_DrawSigilLocations, which determines whether it is appropriate to draw icons on the screen, and if so, what game entities it will point to. I will explain the body of this function in parts, so that you can follow along without getting lost. Start by placing the function opener on line 2476 of cg_draw.c, just after the function CG_DrawWarmup:

```
static void CG_DrawSigilLocations( void ) {
    snapshot_t    *snap;
    int            i;
    vec3_t        origin, end;
    int            redSigil, blueSigil, whiteSigil;

    if ( cgs.gametype != GT_DTF)
        return;

    if ( cg.snap->ps.persistant[PERS_TEAM] == TEAM_SPECTATOR )
        return;
```

In this first bit of code, some local variables are declared, including snap, a pointer to a data type named *snapshot_t*. As discussed in Chapter 5, when a server is in the process of sending information to clients, it wraps pertinent info into a *snapshot*, which are sent at regular intervals. That way, a client can attempt to synchronize the localized information to which the snapshot refers. Typically, this is used for prediction purposes; based on various criteria, a programmer may want to look at the current snapshot of the game, as opposed to the predicted next snapshot. In CG_DrawSigilLocations, similar logic will be used, as you will see shortly.

As the function begins, various tests are performed to see whether the game is in a valid state to draw the flag locator. Because it will checkonly for the *DTF* sigils, a check is performed on the cgs.gametype variable, comparing it to GT_DTF, and returning if they are not equal. Additionally, if the player's team is that of a spectator, the function should also not be drawn.

The next bit of code refers to the snap variable mentioned above:

```
if ( cg.nextSnap && (!cg.nextFrameTeleport && !cg.thisFrameTeleport))
    snap = cg.nextSnap;
else
    snap = cg.snap;

 VectorCopy(cg.snap->ps.origin,origin);

 redSigil = ITEM_INDEX( BG_FindItemForPowerup( PW_SIGILRED ) );
 blueSigil = ITEM_INDEX( BG_FindItemForPowerup( PW_SIGILBLUE ) );
 whiteSigil = ITEM_INDEX( BG_FindItemForPowerup( PW_SIGILWHITE ) );
```

This bit of code tells the *Q3* code the following: "If the player has a valid predicted state, and did not teleport or move a great distance, then use the predicted state of the player; otherwise, use the current state." This simply gives the flag locator a better chance at guessing where the flags will be, based on how the player is moving. The VectorCopy line then begins the process by copying the player's current position into the variable origin.

The variables redSigil, blueSigil, and whiteSigil are assigned values returned from BG_FindItemForPowerup, using their respective powerup_t values as the input parameters. The final result is then wrapped in the ITEM_INDEX macro, which ultimately returns the modelindex for each item. This is important, as you will see with the next bit of code:

```
for ( i = 0; i < snap->numEntities; i++ )
{
    centity_t *target = &cg_entities[snap->entities[i].number];

    if (target->currentState.eType != ET_ITEM)
        continue;

    if ( target->currentState.modelindex != redSigil
        && target->currentState.modelindex != blueSigil
        && target->currentState.modelindex != whiteSigil )
        continue;
```

In this part of the function, a loop begins, looking at each of the entities available to the cgame code. The temporary centity_t variable target is used to examine each entity as the loop iterates. First, the target's currentState.eType value is examined to see if it equals

ET_ITEM. ET_ITEM is defined within the enum entityType_t, and is one of the precious few values communicated from game to cgame.

Unfortunately, due to the simplistic way in which the sigils were implemented in this mod, the only way to determine whether the items being looked at are, in fact, sigils, you must refer to their modelindex. Each target->currentState.modelindex value is checked see whether it matches any of the redSigil, blueSigil, or whiteSigil variables (which, as you know, carry the appropriate modelindex numerical values thanks to the assignment made earlier in the function). If no match is found, the loop iterates to the next entity.

In the lucky event that there is a match, however, the following logic will execute, wrapping up the function:

```
        VectorCopy(target->lerpOrigin,end);
        if (target->currentState.modelindex == redSigil)
            CG_DrawSigilLocationInfo(origin, end,
cgs.media.redFlagShader[0], colorRed);
        else if (target->currentState.modelindex == blueSigil)
            CG_DrawSigilLocationInfo(origin, end,
cgs.media.blueFlagShader[0], colorBlue);
        else if (target->currentState.modelindex == whiteSigil)
            CG_DrawSigilLocationInfo(origin, end,
cgs.media.sigilShader, colorWhite);
    }
}
```

Because you put the player's location in the origin variable, and you want to draw an imaginary line from the player to the sigil, the sigil's location is placed into the end variable with a call to VectorCopy. Then a final if-then-else block is performed, passing the appropriate sigil shader to a new function called CG_DrawSigilLocationInfo. You should recognize the shaders as the same ones used to draw the sigil status HUD update earlier in this chapter. Notice also that each call to the new function adds a final color parameter.

The Quick-and-Dirty CG_DrawSigilLocationInfo

The guts of CG_DrawSigilLocationInfo are what make the flag locator work. Within this function, all that crazy trigonometry you learned in high school comes into play. (Don't feel bad if your trig is a little rough; I don't remember much of anything from high school, either.) For this function, I will try to stick to describing what syntax is used and leave it up to you to research why specific trigonometric functions are employed. (After all, this is *Q3* Programming, not Math 101.)

The goal of this function can be described as follows: you want to be able to draw an imaginary, invisible line between a starting position (the player) and an ending position (a sigil), and then draw an icon on the screen that points in that same direction. As the player rotates, so too will the icon, much like a compass pointing north. You will also want to show a numerical range indicating how far the player is to the target. This will probably be the single most complicated function in the entire book, so I'll break it down piece by piece.

Start by opening the new function above `CG_DrawSigilLocations`, using the following snippet:

```
void CG_DrawSigilLocationInfo( vec3_t origin, vec3_t target, qhandle_t
shader, vec4_t color )
{
    int      x = 320, y = 240;
    int      w = 320, h = 240;
    float    angle, distance;
    vec3_t   temp, angles;

    VectorSubtract(origin, target, temp);
    distance=VectorLength(temp);
```

`CG_DrawSigilLocationInfo` opens by declaring variables `x` and `y` to hold the center point of the screen (recall that in *Q3*, all coordinates are based on 640 × 480, regardless of screen resolution), and variables `w` and `h` to hold half the screen's dimensions. In both cases, the values are 320 × 240. The first task executed in this function is to determine the distance between the origin in the target, which is performed by calling `VectorSubtract` using `origin` and `target` as the parameters, and saving the result to `temp`. The `temp` vec3_t is then passed to `VectorLength` to return a distance.

```
    VectorNormalize(temp);
    vectoangles(temp,angles);

    angles[YAW]=AngleSubtract(cg.snap->ps.viewangles[YAW],angles[YAW]);
```

This next bit of code passes the temp vec3_t (which was the vector holding the difference between origin and target) to VectorNormalize, rounding the distance out to 1. That normalized vector is then passed to vectoangles, creating an angle of the vector, to be held in angles. As declared previously, angles is of data type vec3_t, but in this instance, it is really used as a three-dimensional array to hold three types of angles: pitch, yaw, and roll. *Pitch* is the angle of the player looking up or down, whereas *yaw* is the angle of the player looking left or right. *Roll* refers to the degree that you are tilted over (to the left or right.) Most often, roll is modified when a player has fallen over on his side after being killed. The angle type you care about here is yaw.

The next line of code shows a call to AngleSubtract, which determines the difference between them, resulting in a value that ranges anywhere from –180 to 180. In this usage of the function, AngleSubtract assists you in converting the base angle in the original vector to the relative angle of the player. The result of this difference is committed back to angles[YAW].

```
angle=(angles[YAW] + 180.0f)/360.0f;
angle -=0.25;
angle *= (2*M_PI);
```

These next three lines use the variable angle, which is a float. First, the yaw of angles is converted to radians by adding 180.0 and then dividing the result by 360, returning the result to angle. Next, 0.25 is subtracted from angle; this is done because the *Q3* coordinate system starts at –90 degrees, whereas radians start at 0. Subtracting 0.25 from angle has the effect of rotating the angle a quarter turn. angle is then multiplied by pi times two (3.14159 * 2).

```
w=sqrt((w*w)+(h*h));
x +=cos(angle)*w;
y +=sin(angle)*w;
```

In these next three lines, the w variable (currently holding half the width of the screen, 320) is passed to the sqrt (square root) function, using width^2 plus height^2 as the value. The The result of this square root is returned to w. Next, the center point of the screen (held in x and y) is offset by using cos (cosine) and sin (sine) trigonometric functions, passing angle in as the input parameter, and multiplying the result by the new w variable.

The resulting coordinates held in x and y represent circular positioning on the HUD. The last time I checked, however, the HUD was a square, so certain adjustments are made to x and y so that the circular coordinates are made to fit inside the square HUD:

```
if (x<15)
    x=15;
else {
    if (x>605)
        x=605;
}

if (y<20)
    y=20;
else {
    if (y>440)
        y=440;
}
```

Very simply, if x is less than 15 or greater than 605, it is capped back to each (respectively). In the same manner, if y is less than 20 or greater than 440, it is also constrained. Finally, with the proper x and y location ready for plotting to the HUD, the function concludes by drawing the HUD icons and a numerical distance, like so:

```
    CG_DrawPic( x, y, 20, 20, shader );
    CG_DrawStringExt( x-50, y+20, va("%10.2f",distance/100.0), color,
qtrue, qfalse, TINYCHAR_WIDTH, TINYCHAR_HEIGHT, 0 );
}
```

As you can see, shader is used in the call to CG_DrawPic, as it was originally passed to CG_DrawSigilLocationInfo. In the CG_DrawStringExt call, the coordinates are offset slightly to render the text below the shader icons and slightly to the left (to give a more centered appearance). The old distance variable, which was used earlier in the function to determine the length from the origin to the target, is used here as the actual text to be displayed on screen (it is abbreviated somewhat by dividing the value by 100).

The variable color specifies the color of the text, as it was passed from CG_DrawSigilLocations. The next qtrue parameter tells CG_DrawStringExt to force that color to be used, while the following qfalse parameter opts out of drawing a drop shadow behind the text. Finally, TINYCHAR_WIDTH and TINYCHAR_HEIGHT are used to reference the dimensions of the text, translating to 8 × 8 pixels, and the final value, 0, indicates that there is no maximum number of characters to be displayed.

Phew! If you made it through that last function and can still feel your legs, you've survived a hefty math lesson! As you get into more exciting and innovative code, you will undoubtedly be using more trigonometry and physics, so it might not hurt to read up on those subjects.

Now that you have all the necessary code in place to generate the flag locator, let's make one final addition to actually bring this new addition to life. Hop all the way down to line 2656, putting you deep into CG_Draw2D, the function responsible for drawing two-dimensional effects on the HUD. After the call to CG_DrawLagometer, check the integer member of your new Cvar to see if it is 1. If so, make a call to the flag-locator function:

```
CG_DrawLagometer();

if (cg_sigilLocator.integer == 1)
    CG_DrawSigilLocations();
```

Sigil to Player: I'm Over Here!

I know you're eager to build your DLLs and test the flag locator out, but you need to make one more quick change. Currently, sigils act like idle items in *Q3*; the game doesn't do anything special to announce their presence. Players simply find them, touch them, and carry on playing as usual. However, the flag locator will constantly need to know where the sigils are, regardless of their distance from the player. Thus you need a way to indicate to *Q3* that sigils are to announce their presence, so that functions such as the flag locator can pick up their signal. You'll do this by adding the flag SVC_BROADCAST to the sigil's entityShared_t member svFlags.

The first two sigils are created during G_CallSpawn, so open g_spawn.c and scroll to line 284. After ent->classname is assigned to item->classname, add the SVC_BROADCAST flag, like so:

```
ent->classname = item->classname;
ent->r.svFlags = SVF_BROADCAST;
```

The third sigil is generated dynamically, in `ValidateSigilsInMap`. That function is in g_team.c, so open that file, hop down to line 279, and after `targ->item` is assigned to `item`, make the same flag assignment:

```
        targ->item = item;
        targ->r.svFlags = SVF_BROADCAST;
```

With that, you can wipe the sweat from your furrowed brow; the flag locator is complete. Go ahead and build the qagamex86.dll and cgamex86.dll, drop them in your MyMod folder, and fire up *Q3* with this command line:

```
quake3.exe +set fs_game MyMod +set sv_pure 0 +set g_gametype 5 +set
cg_sigilLocator 1 +map q3ctf1
```

You should see flag icons somewhere around the perimeter of your screen, indicating the color of each flag as well as the distance to each flag. The location of each icon gives you a general direction you can start heading toward (see Figure 10.2).

This addition to *DTF* is extremely cool. The new flag locator will act as a compass and help new players find flags in maps that they haven't

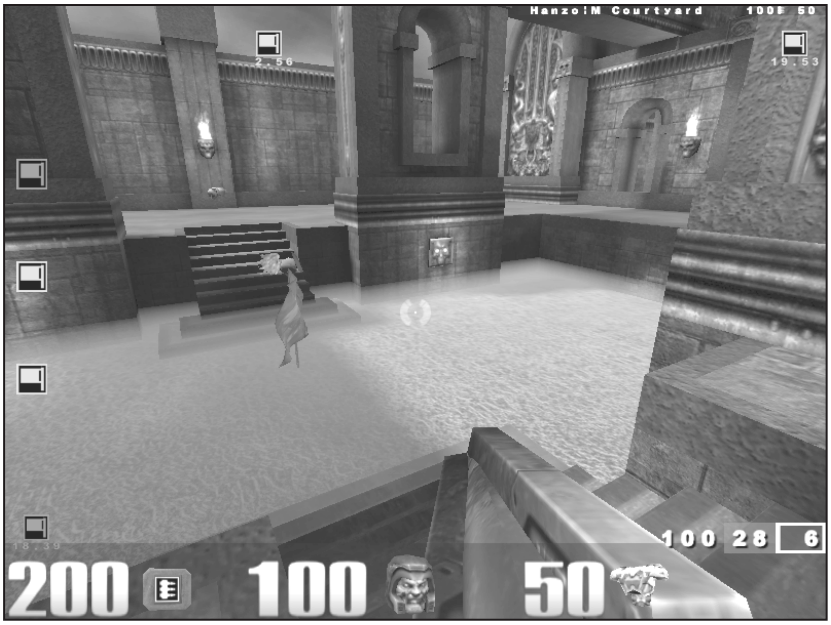


Figure 10.2 *The newly added flag locator in* DTF

played in before, as well as assist experienced players with finding the dynamically generated third point in pre-existing *CTF* maps. And, because you implemented it using a Cvar, players are free to turn the flag locator off if it gets too distracting, by typing `cg_sigilLocator 0` in the console.

Adding *DTF* to the UI

The final bit of polish you can apply to your new *DTF* mod is an addition to the user interface. *DTF* reuses existing *CTF* maps, so you'll want to be able to allow players to select any of the *CTF* maps when they create a new game from the menu system. As well, you will want to allow players to enable or disable new options in a game of *DTF*, such as the recently implemented flag locator. In this section, you'll revisit the `ui` code and make adjustments so that players can quickly and easily set up a game of *DTF* from the in-game menu.

Specifying the Setup of *DTF*

In order to build *DTF* support into the *Q3* user interface, you should first list any items that players must specify at startup. For example, when a player starts a new game server in *Q3*, he will enter the multiplayer menu and then click on the Create button to view the Game Server menu (see Figure 10.3).

From here, the player must be able to cycle through the available game types and choose *Defend the Flag* from the list. When the player clicks the Next button, he will see a new screen that allows him to specify certain game-related info, such as the time limit, capture limit, server name, and so forth. You should, however, give him two new options on this page. The first option will be to enable or disable the flag locator that you just created. As cool as you and I might think the flag locator is, some players may be annoyed by it or find it too distracting, so it's always a good idea to give players the ability to disable a feature you've added.

The second new option you'll allow a player to modify on this page deals with spawn points. Typically, *CTF* spawn points are segregated to each team. Red players spawn in and around the red base, while blue players spawn in and around the blue base. Because you are using *CTF* maps for *DTF*, these team-based spawn points will carry over.

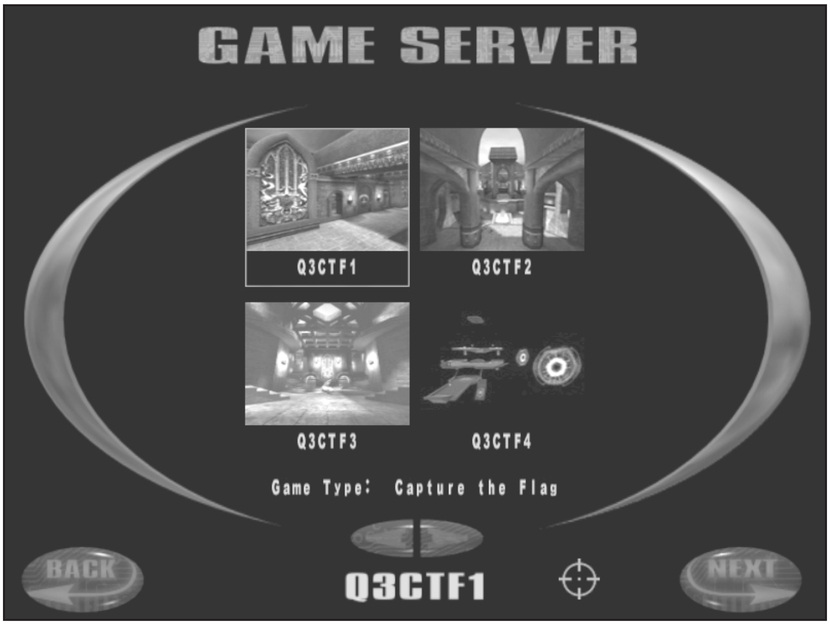


Figure 10.3 *The Game Server menu in* Q3

They may not always be appropriate, however, because the sigils in *DTF* don't really belong to any team the way flags do in *CTF*. The goal of *DTF* is to actively get players to hold all three sigils, and if red players are constantly spawning near what *used to be* the red flag, they pretty much have full control over that sigil for the duration of the game (ditto blue players and their old blue-flag spawn point). It makes more sense to allow players to spawn randomly in the map in a game of *DTF*, using the deathmatch spawn points; therefore, you should offer the option to use *DM* spawning versus *CTF* team-based spawning.

Handling Two Different Spawning Styles

Let's start this update by opening g_local.h and scrolling down to line 747. As discussed in the previous section, this is where the series of Cvars that apply to game code are declared. Go ahead and add a new `extern` declaration of a Cvar, called `g_dtfspawnstyle`, like so:

```
extern vmCvar_t    g_proxMineTimeout;
extern vmCvar_t    g_dtfspawnstyle;  // dtf or ctf spawn points?
Additionally, make its local declaration in g_main.c, on line 77:
vmCvar_t    g_proxMineTimeout;
#endif
vmCvar_t    g_dtfspawnstyle;  // dtf or ctf spawn points?
```

Just like any other Cvar, players can access this variable from the console simply by typing `g_dtfspawnstyle` and then a value. The update you will build to the UI, however, will prevent the user from having to know the specific name of the Cvar.

Now that the Cvar exists, you will need to have it initialized when *Q3* fires up. Open g_main.c and head to line 157, which places you back in the middle of the `gameCvarTable` array declaration. After the element `g_proxMineTimeout` is defined, add an element for `g_dtfspawnstyle`:

```
    { &g_proxMineTimeout, "g_proxMineTimeout", "20000", 0, 0, qfalse },
#endif
    { &g_dtfspawnstyle, "g_dtfspawnstyle", "0", CVAR_SERVERINFO |
CVAR_ARCHIVE | CVAR_NORESTART, 0, qtrue },
```

As you learned in the previous section, this element initializes the Cvar `g_dtfspawnstyle` into the array, with the string `"g_dtfspawnstyle"` (which is what is typed into the console to access it), and it has a default value of `0` set. It also receives the flags `CVAR_SERVERINFO`, `CVAR_ARCHIVE`, and `CVAR_NORESTART`. Finally, its `modificationCount` is reset to `0`, and `trackChange` is set to `true`, making *Q3* announce when the Cvar is updated. Your new Cvar is now in the game; the next step is to see how team spawns are controlled.

Open the g_client.c file and scroll down to line 1049, putting yourself into the body of the function `ClientSpawn`. This is the function that's called whenever a new player is first placed into a map, as well as when a player has died and needs to be re-spawned. On line 1049, you should see the following logic:

```
    } else if (g_gametype.integer >= GT_CTF ) {
        // all base oriented team games use the CTF spawn points
        spawnPoint = SelectCTFSpawnPoint (
                        client->sess.sessionTeam,
                        client->pers.teamState.state,
                        spawn_origin, spawn_angles);
```

This tells you that if the Cvar g_gametype's integer value is greater than or equal to GT_CTF, then a spawn point is generated through a call to the function SelectCTFSpawnPoint. Otherwise, standard spawn generation is used. Numerically, GT_DTF is a higher value than GT_CTF, which means that this function will be called for DTF as well. However, you'll want a *CTF* spawn point only if the new g_dtfspawnstyle Cvar is equal to 1 (and the current game type is *DTF*). To implement that rule, change the preceding code to read like this:

```
        } else if ( (g_gametype.integer == GT_DTF &&
g_dtfspawnstyle.integer == 1) ||
                (g_gametype.integer >= GT_CTF && g_gametype.integer !=
GT_DTF ) ) {
```

Now, if a game of *DTF* is in progress, and g_dtfspawnstyle has a value of 1—or the current game type is GT_CTF or greater (but not GT_DTF)—then a *CTF* spawn point will be used. Otherwise, the standard, random *DM* spawns will be used.

Making *DTF* Selectable

The first part of implementing *DTF* into the *Q3* user interface requires that you establish *Defend the Flag* as a selectable game type. To do this, you must inform the ui of this new game type through the use of initialization. Start by opening ui_startserver.c, which is the file responsible for the Game Server creation menus. Line 76 is where you will find your first bit of code to modify: the declaration of the gametype_items array, which feeds the available game types to the Game Server's menulist_s control. Go ahead and add an entry for *Defend the Flag*, right after the one for *Capture the Flag*.

```
static const char *gametype_items[] = {
        "Free For All",
        "Team Deathmatch",
        "Tournament",
        "Capture the Flag",
        "Defend the Flag",
        0
};
```

This updated list will now populate the Spin control at the bottom of the first Game Server menu screen, allowing players to choose *DTF* as their game type. The gametype_items array is also used in other places to display the game type, such as in the level-shot window found on the second Game Server menu. The next two lines of code in ui_startserver.c are additional array declarations used to map the appropriate game flags (like GT_DTF) to their string descriptions in gametype_items. Go ahead and modify the next two lines so that they read as follows:

```
static int gametype_remap[] = {GT_FFA, GT_TEAM, GT_TOURNAMENT, GT_CTF,
GT_DTF};
static int gametype_remap2[] = {0, 2, 0, 1, 3, 4};
```

Here, you have simply added a GT_DTF element to the end of gametype_remap, and because it falls in the fourth index of that array, the number 4 is added to the end of gametype_remap2. So, when the description of the game type is required, such as is the case in the final line of code in ServerOptions_LevelshotDraw:

```
UI_DrawString( x, y,
gametype_items[gametype_remap2[s_serveroptions.gametype]],
UI_CENTER|UI_SMALLFONT, color_orange );
```

the string can be obtained by pulling from the index of gametype_remap2 that equals the current s_serveroptions.gametype value—which should equal GT_DTF. Because GT_DTF really equals 5, the fifth element in gametype_remap2 equals 4, mapping back to "Defend the Flag" in gametype_items.

CTF Maps are OK in My Book

It's no secret at this point that your awesome coding ability allows *Q3* to reuse *CTF* maps in *DTF*, dynamically changing flag points into sigil points, and so on. The UI, however, doesn't know that. When a player selects *DTF* as his game type, you will want to indicate to the UI that *CTF* maps are still applicable and should be offered to the player as a possible location for his next battle. *Q3* determines which maps are appropriate for the specified game type in a function called StartServer_GametypeEvent. In this function (found on line 226 of ui_startserver.c), bit flags for the current game are generated off of the gametype_remap array (I told you it was used!), using the current value of the Spin control (representing the list of games) as the index:

```
matchbits = 1 << gametype_remap[s_startserver.gametype.curvalue];
```

Then, another set of bit flags are generated, based on the type of map found during a parse of all available maps in *Q3*:

```
gamebits = GametypeBits( Info_ValueForKey( info, "type") );
```

Some examples of what a map's `"type"` could be include `"team"`, `"single"`, `"tourney"`, and `"ctf"`. The call to `GametypeBits` performs the physical generation of the bit flags based on the map's `"type"`, so let's jump to line 130 in ui_startserver.c, which should put you at the end of this function. The last comparison of the map's `"type"` is to that of `"ctf"`:

```
        if( Q_stricmp( token, "ctf" ) == 0 ) {
            bits |= 1 << GT_CTF;
            continue;
        }
```

Because all *CTF* maps will also be playable in *DTF*, indicating this fact to *Q3* is as simple as adding an additional bit flag, bit-shifted off of `GT_DTF`'s value:

```
        if( Q_stricmp( token, "ctf" ) == 0 ) {
            bits |= 1 << GT_CTF;
            bits |= 1 << GT_DTF; // dtf can play ctf maps
            continue;
        }
```

Now, when a user selects *Defend the Flag* at the bottom of the first Game Server menu, only *CTF* maps should be listed and selectable.

Adding *DTF* Options to the Game Server Menu

The first page of the Game Server menu is complete. Players can now successfully cycle through the list of game types during the creation of a new game and specify *Defend the Flag* as their game of choice. Also, by selecting *DTF* they will be given a proper list of *CTF* maps from which to choose. The final adjustment you'll need to make is to modify the second Game Server screen, which will allow players to alter *DTF*-specific options, including the flag locator and spawn style.

Because you're an expert in `ui` modification by now, you should know that the first change will be to add two new controls to the serveroptions_t struct, the data type of `s_serveroptions`, which controls the second Game Server menu. Jump down to line 617, which puts you into the declaration of serveroptions_t. After `hostname`, add a menulist_s control and a menuradiobutton_s control, like so:

```
    menufield_s           hostname;
    menulist_s            dtfspawnstyle; // dtf spawn style
    menuradiobutton_s    sigillocator;  // dtf sigil locator
```

The new variables `dtfspawnstyle` and `sigillocator` will house your two additional controls used for *DTF* games. Because `dtfspawnstyle` is a menulist_s, you'll also want a const char array of possible values through which the control can cycle. Hop down to line 673 after the other const char arrays for the Game Server menu are declared, and add a new one after `botSkill_list`:

```
    "Nightmare!",
    0
};

// for dtfspawnstyle
static const char *dtfspawn_list[] = {
    "DM Spawns",
    "CTF Team Spawns",
    0
};
```

Here, the const char array simply holds two values: "DM Spawns", which will be displayed when the `g_dtfspawnstyle` Cvar is 0, and "CTF Team Spawns" when its value is 1.

The next step is to initialize these new controls. To do that, jump down to line 1250. Here, the function ServerOptions_MenuInit is in the process of initializing all the controls used for the second Game Server menu. The first change you'll want to make is to ensure that `capturelimit` is an available score type in *DTF,* as it is in *CTF.* On line 1250, you should see the following code:

```
if( s_serveroptions.gametype != GT_CTF ) {
```

This indicates that *Q3* will use `fraglimit` as an available score type, as opposed to `capturelimit`, if the game is not `GT_CTF`. To add `GT_DTF` to this exclusion, modify the code as follows:

```
if( s_serveroptions.gametype != GT_CTF && s_serveroptions.gametype !=
GT_DTF ) {
```

Success! `capturelimit` is now the scoring typing for *DTF* as well as *CTF*.

The last control on the page is the Hostname control, used to identify the name of the game server being created. The two new *DTF* controls can be placed under Hostname, so on line 1320, after the `hostname` variable is initialized, add the following code:

```
    if (s_serveroptions.gametype == GT_DTF) {
        y += BIGCHAR_HEIGHT+2;
        s_serveroptions.dtfspawnstyle.generic.type         =
MTYPE_SPINCONTROL;
        s_serveroptions.dtfspawnstyle.generic.flags        =
QMF_PULSEIFFOCUS|QMF_SMALLFONT;
        s_serveroptions.dtfspawnstyle.generic.x            =
OPTIONS_X;
        s_serveroptions.dtfspawnstyle.generic.y            =
y;
        s_serveroptions.dtfspawnstyle.generic.name         =
"Spawn Style:";
        s_serveroptions.dtfspawnstyle.itemnames            =
dtfspawn_list;

        y += BIGCHAR_HEIGHT+2;
        s_serveroptions.sigillocator.generic.type          =
MTYPE_RADIOBUTTON;
        s_serveroptions.sigillocator.generic.flags         =
QMF_PULSEIFFOCUS|QMF_SMALLFONT;
        s_serveroptions.sigillocator.generic.x             =
OPTIONS_X;
        s_serveroptions.sigillocator.generic.y             =
y;
        s_serveroptions.sigillocator.generic.name          =
"Flag Locator:";
    }
```

An item worthy of mention in this code snippet is the lack of `generic.id` and `generic.callback` assignments for each control. This is done for a reason, which I will get into shortly. Note that each control has its appropriate `generic.type` set, as well as its `generic.name` label.

The `dtfspawnstyle` control also has its const char array assigned to it, as indicated by `generic.itemnames` being set to `dtfspawn_list`.

Don't forget to add the new controls to the menu. Hop down further to line around 1456, and after the `hostname` control is added, make the following adjustments:

```
        Menu_AddItem( &s_serveroptions.menu, &s_serveroptions.hostname );
    }

    if (s_serveroptions.gametype == GT_DTF)
    {
        Menu_AddItem( &s_serveroptions.menu, &s_serveroptions.
dtfspawnstyle );
        Menu_AddItem( &s_serveroptions.menu, &s_serveroptions.
sigillocator );
    }
```

You'll also want to make sure that the `capturelimit` control is added in *DTF*, because you previously specified in the initialization that you wanted `capturelimit` in a *DTF* game. That code occurs up near line 1438, and should be modified to read as follows:

```
     if( s_serveroptions.gametype != GT_CTF && s_serveroptions.gametype
!= GT_DTF ) {
         Menu_AddItem( &s_serveroptions.menu, &s_serveroptions.
fraglimit );
     }
```

Before, the check was simply made to see if `s_serveroptions.gametype` was not equal to `GT_CTF`; your addition will exclude `GT_DTF` as well (the result is that neither *CTF* nor *DTF* will gain the `fraglimit` control).

The last bit of code modification will involve actually setting the values of the new controls to the game being created. As I mentioned earlier, the `generic.id` and `generic.callback` functions were not specified for the new controls (nor did you define an `ID_` variable to identify each control). Back in Chapter 9, it was made painfully clear that if your controls were to accept user input and change variables in the game, they would need to call some kind of function, identifying themselves in the process. How, then, will these new controls work? The answer lies within `ServerOptions_Start`.

The Game Server menu is set up a little differently from other menus you've worked with, in that it doesn't commit any selections until the game is told to start. There may be many reasons why the Game Server menu was designed this way, probably for ease of development. Regardless, all the game-related variables that are configured during setup are saved in `ServerOptions_Start` when the user launches the new server. To add your new controls to this function, scroll to line 717, where various local variables are declared for use in this function. After `flaglimit`, add two integers to represent the values of your new controls:

```
    int         flaglimit;
    int         dtfspawnstyle; // dtf
    int         sigillocator;  // dtf
```

Next, you should see that, a few lines down, the majority of those variables are set by polling the current values of all the controls in the Game Server menu. After the skill variable is assigned, you must add assignments for your new integers, looking at each of the new controls' `curvalue`:

```
    skill          = s_serveroptions.botSkill.curvalue + 1;
    dtfspawnstyle = s_serveroptions.dtfspawnstyle.curvalue; // dtf
    sigillocator  = s_serveroptions.sigillocator.curvalue;  // dtf
```

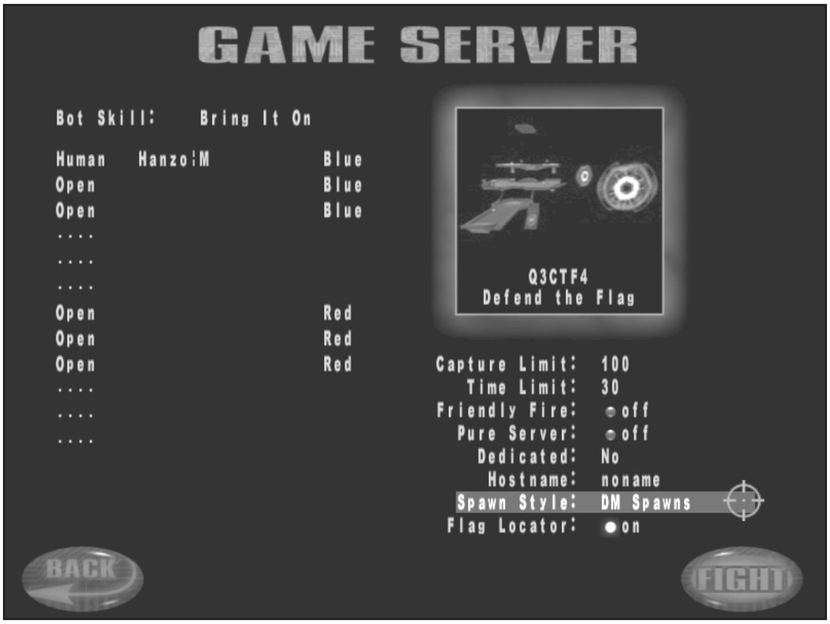
Nothing too complicated is going on here; you're simply looking at the current value of each control and assigning it to the new integer variables you declared above. Your final (and I mean final!) task is to commit the values of those integers to the actual Cvars that control the matching logic in `game` and `cgame`. Scroll down to line 776, and look at the series of function calls being made. The system-call function `trap_Cvar_SetValue` is called in succession, assigning the value of each local integer to an appropriate Cvar. Take a deep breath and make your final adjustment, adding assignments for `g_dtfspawnstyle` and `cg_sigilLocation`, right after `capturelimit`:

```
    trap_Cvar_SetValue ("capturelimit", Com_Clamp( 0, flaglimit,
flaglimit ) );
    trap_Cvar_SetValue ("g_dtfspawnstyle", Com_Clamp( 0, dtfspawnstyle,
dtfspawnstyle ) );
    trap_Cvar_SetValue ("cg_sigilLocator", Com_Clamp( 0, sigillocator,
sigillocator) );
```

TIP

In the code that sets the Cvars, you may have noticed a call to a function called `Com_Clamp`. `Com_Clamp` is used to specify a minimum and maximum value (parameters one and two), so that a third parameter does not exceed either.

Hooray! Your modification to the `UI` to support *DTF* is complete! You're now free to build your uix86.dll and qagamex86.dll, drop them in your MyMod folder, and fire up the new menu system. Select the Multiplayer menu, click the Create button, and cycle through the game types until *Defend the Flag* is displayed. Select a map, and click Next. You should see the new controls listed near the bottom-right corner of the screen (see Figure 10.4).



```
Bot Skill:    Bring It On

Human   Hanzo¦M          Blue
Open                     Blue
Open                     Blue
....
....
....
Open                     Red
Open                     Red
Open                     Red
....
....
....

Q3CTF4
Defend the Flag

Capture Limit:  100
   Time Limit:  30
Friendly Fire:  off
  Pure Server:  off
    Dedicated:  No
     Hostname:  noname
  Spawn Style:  DM Spawns
 Flag Locator:  on
```

Figure 10.4 *The Game Server menu with additional* DTF *controls*

Summary

In this chapter, you took steps to improve upon the existing *DTF* modification. By creating a new flag-status indicator on the HUD, as well as a flag locator, you have further increased the value and dynamics of this modification, refining the game type and building on the strengths of its new rules. Players who will be running through already familiar *CTF* maps will greatly benefit from the indicator that identifies which flags are held by which team, and the flag locator will assist them in tracking new flags down. As well, adding the *DTF* game type to the user interface makes it easier for players to quickly get in and set up new games, without having to read up on specific rules or names of Cvars. As great a tool as documentation is, some gamers prefer to get in as fast as possible; making adjustments to the `ui` code to support *DTF* only makes it easier for players in the long run.