

THE PREMIER PRESS

GAME DEVELOPMENT SERIES

CD INCLUDED



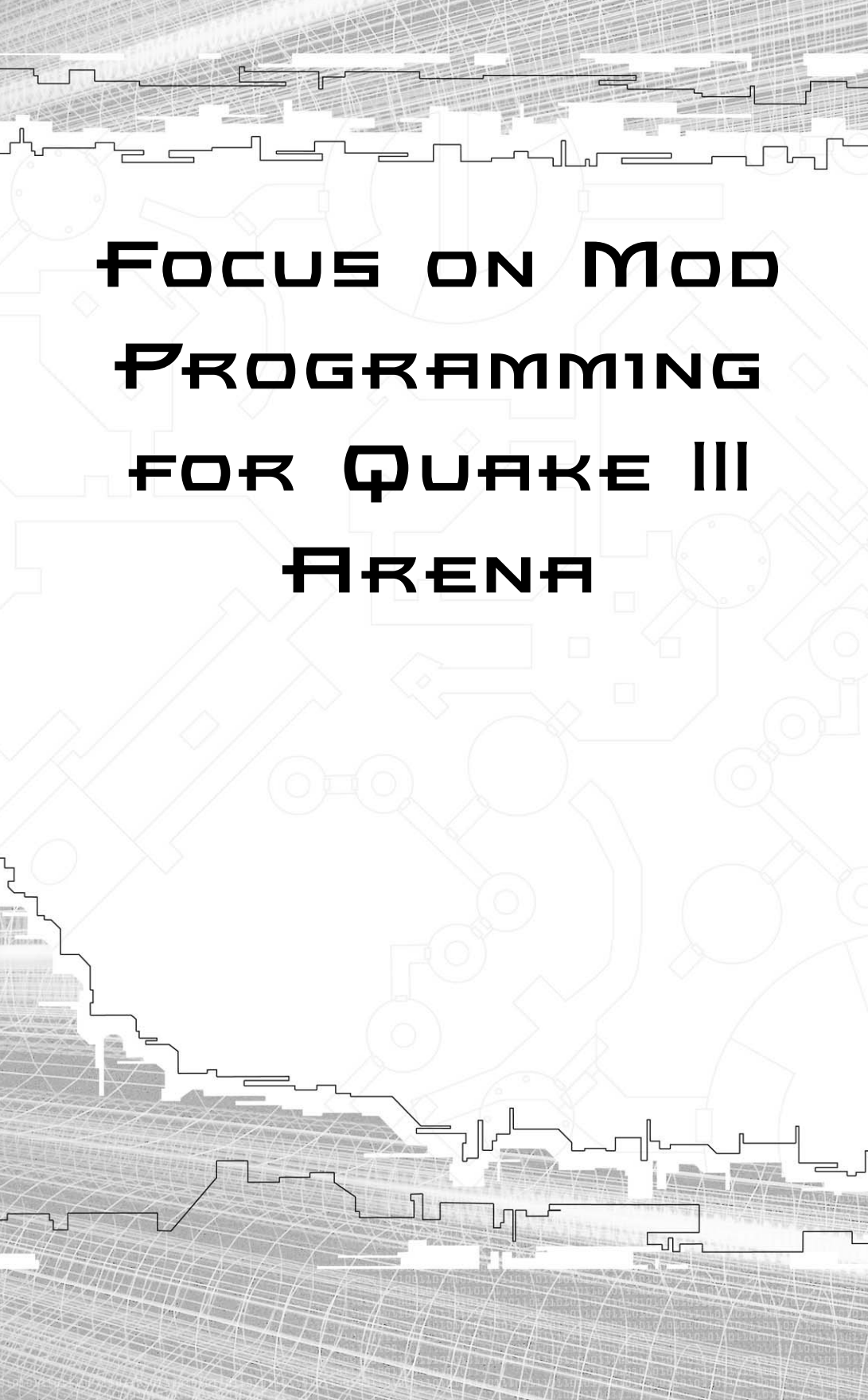
**FOCUS ON**

# MOD PROGRAMMING IN QUAKE III ARENA

Shawn Holmes

Series Editor  
André LaMothe  
CEO Xtreme Games LLC





**FOCUS ON MOD  
PROGRAMMING  
FOR QUAKE III  
ARENA**

*This page intentionally left blank*

# FOCUS ON MOD PROGRAMMING FOR QUAKE III ARENA

**Shawn Holmes**

PREMIER PRESS



GAME DEVELOPMENT

Premier



Press

© 2002 by Premier Press, a division of Course Technology. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system without written permission from Premier Press, except for the inclusion of brief quotations in a review.



The Premier Press logo, top edge printing, and related trade dress are trademarks of Premier Press, Inc. and may not be used without written permission. All other trademarks are the property of their respective owners.

**Publisher:** Stacy L. Hiquet  
**Marketing Manager:** Heather Hurley  
**Managing Editor:** Sandy Doell  
**Acquisitions Editor:** Emi Smith  
**Series Editor:** André LaMothe  
**Project Editor:** Estelle Manticas  
**Editorial Assistant:** Margaret Bauer  
**Technical Reviewer:** Robi Sen  
**Technical Consultant:** Jared Larson  
**Copy Editor:** Kate Welsh  
**Interior Layout:** Marian Hartsough  
**Cover Design:** Mike Tanamachi  
**Indexer:** Katherine Stimson  
**Proofreader:** Jennifer Davidson

All trademarks are the property of their respective owners.

*Important:* Premier Press cannot provide software support. Please contact the appropriate software manufacturer's technical support line or Web site for assistance.

Premier Press and the author have attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

Information contained in this book has been obtained by Premier Press from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, Premier Press, or others, the Publisher does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from use of such information. Readers should be particularly aware of the fact that the Internet is an ever-changing entity. Some facts may have changed since this book went to press.

ISBN: 1-931841-56-X

Library of Congress Catalog Card Number: 2001099836

Printed in the United States of America

02 03 04 05 RI 10 9 8 7 6 5 4 3 2 1

Premier Press, a division of Course Technology  
2645 Erie Avenue, Suite 41  
Cincinnati, Ohio 45208

*To Ariel and Hunter,  
who love to play games  
almost as much as their Dad*

*And to Dave Kirsch,  
for inspiring me, and many others,  
to create*

# Acknowledgements

**T**his is my first book, and it was both challenging and a constant learning process from start to finish. Because of the unusual circumstances that led to my becoming the author, I have many important people to thank; without them I surely wouldn't have been able to pull this off. The first of these people are Emi Smith and Estelle Manticas, two patient ladies who held my hand as I stumbled blindly.

Many of the tutorials in this book are based on the hard work of dedicated *Quake III Arena* fans, who took their own personal time to write up what they had discovered. My thanks go to Chris Hilton, Karl Pauls, Arthur "Calrathan" Tomlin, and Ian "HypoThermia" Jefferies at *Code3Arena* for their extensive help and knowledge. As well, thanks to my favorite 3D modeler and friend Dave Wolfe for his contribution in the modeling department. My passion for C programming would not exist if it weren't for the support of Dave Sausville, and so I have him to thank as well. I need to thank Thaxton Beesely, too, for it was his laptop on which a great deal of this book was written.

Special thanks go to Robi Sen, my technical editor, for believing in me enough to get me on this project. Jared Larsen provided detailed information on *Q3*'s code, shaders, and wrote the Flag Indicator & Dynamic Spawn generating functions (all after having just welcomed a baby!), so thanks very much to him. Many thanks also to Andy Smith for providing introductory content in Chapter 2. Also, special thanks to Anthony Jacques, for providing the source to his "Domination" mod for *Q3*, which formed the basis of the *Defend the Flag* modification, handled in the last section of this book. Thanks also go to the team back at Breckenridge Communications, who dared me to write a book.

And finally, thanks especially to my wife, Julie, for her support and patience during this exciting and unpredictable project.

# About the Author

**SHAWN HOLMES** is a gaming addict, period. Born in the recesses of the Great White North, Shawn grew up in the small town of Parksville, British Columbia, with dreams of one day becoming a professional video-game developer. He first tinkered with an Apple computer back in 1985, and wrote his first game for a Computer Science project in 1990. Today, Shawn is Senior Programmer at Breckenridge Communications, an Internet-applications company in Denver, Colorado, and lives with his wife Julie and their two children, Ariel and Hunter. He can be reached at [sholmes@breckcomm.com](mailto:sholmes@breckcomm.com).

Shawn has played a role in several game modifications; in 1997 he headed up *The CTF Expansion Project* to increase the number of quality CTF maps available for *Quake*. In 1999, his *Decay!* mod for *Heretic II* was named "Mod of the Month" by an Australian gaming magazine, *PC Powerplay*. As long as video and computer games exist, Shawn will continue to push his development experience to new levels in his quest to create the next exciting game.



# Contents

LETTER FROM THE SERIES EDITOR . . . . . XIV

INTRODUCTION . . . . . XVII

## CHAPTER 1 INTRODUCTION TO PROGRAMMING MODS . . . . . 1

What Is a Mod? . . . . .	2
Why Create a Mod Instead of Just Writing a Game? . . . . .	3
The Tools of the Trade . . . . .	4
Using C . . . . .	4
Using the VC++ Compiler . . . . .	5
What Mods Are Currently Available? . . . . .	6
Urban Terror . . . . .	6
Rocket Arena Q3A . . . . .	7
Quake III Fortress (Q3F) . . . . .	8
Painkeep Arena . . . . .	9
Rune Quake . . . . .	11
Summary . . . . .	12

## CHAPTER 2 C PROGRAMMING IN QUAKE III . . . . . 13

The History of Quake and Its Code . . . . .	15
The Move from DOS to Win32 . . . . .	16
Hello Quake III, Goodbye DLL! (Sort Of) . . . . .	18
Getting Set Up . . . . .	19
Installing Q3 and the Source . . . . .	19
The Source Directory . . . . .	21
Using Visual C++ . . . . .	22

Building the Source . . . . . 25

    Compiling the Project. . . . . 27

    Loading Up Your Mod . . . . . 28

Looking at the Quake III Code . . . . . 29

    On Your Marks, Get Set . . . . . 29

    The Q3 Source. . . . . 30

    Planning . . . . . 33

A Simple Mod: The Homing Missile . . . . . 33

    I Think, Therefore I NextThink . . . . . 36

    Entities: Building Blocks in Q3. . . . . 37

    Changing the Missile's Behavior. . . . . 38

    Smoothing the Missile. . . . . 47

    A Final Note. . . . . 48

Summary . . . . . 48

**CHAPTER 3 MORE WEAPONRY WORK. . . . . 49**

Understanding Weapon Types . . . . . 50

Modifying the Shotgun . . . . . 52

    Understanding the Top-Down Approach . . . . . 52

    Knowing the Shotgun Inside and Out . . . . . 53

    The Physics of Vectors . . . . . 55

    Intricacies of Damage . . . . . 56

    Adjusting the Shotgun's Accuracy . . . . . 58

    The Shotgun's Dirty Secret. . . . . 62

    Synchronicity in the Client Code . . . . . 66

    Adding Polish: Shooting While Moving. . . . . 68

Modifying Grenades: The Cluster Grenade. . . . . 72

    Further g.weapon.c Detective Work . . . . . 73

    Why a Grenade Bounces (and Rockets Don't) . . . . . 74

    Using What You Know: think and nextthink. . . . . 76

    Making the Cluster Grenade Behave. . . . . 78

A Further Adjusting Gravity Wells . . . . . 82  
 Into the Vortex . . . . . 83  
 Testing for Collision . . . . . 89  
 Making Gravity Work for You . . . . . 90  
 Summary . . . . . 91

**CHAPTER 4 MANIPULATING  
 THE PLAYER . . . . . 93**

The Quake III Player and His World. . . . . 94  
 Player Structure . . . . . 96  
 The Guts of gclient\_s . . . . . 97  
 State of the Player. . . . . 102  
 Changing the Player’s Movement . . . . . 105  
 Playing with ps.speed. . . . . 106  
 Gravity Kills . . . . . 107  
 The Case of the Missing client . . . . . 108  
 Solving the Jumping Mystery . . . . . 109  
 The Move to Pmove . . . . . 112  
 Modifying Jump Velocity. . . . . 114  
 Giving the Player a Jetpack. . . . . 115  
 Creating a New pmove Flag . . . . . 115  
 Defying Gravity . . . . . 117  
 A Surprise Effect. . . . . 118  
 Borrowing Code from PM\_FlyMove . . . . . 119  
 Implementing Locational Damage. . . . . 124  
 Creating Hit Flags . . . . . 124  
 The Bounding Box . . . . . 126  
 Scanning Body Parts . . . . . 128  
 My Feet Are . . . Where Now? . . . . . 130  
 Switching Off the Hits. . . . . 131  
 Summary . . . . . 132

**CHAPTER 5 QUAKE  
COMMUNICATION . . . . . 133**

The Client/Server Relationship . . . . . 134

    Leaving Peering in the Past . . . . . 135

    Lag in a Nutshell . . . . . 138

    The Quest for Low Latency . . . . . 139

    The Process of Updates . . . . . 140

Bridging game, cgame, and ui . . . . . 143

    The Server Is God . . . . . 144

    The Main Event . . . . . 145

    Interfacing with the User . . . . . 147

    Modifying Variables on-the-Fly . . . . . 149

The Quake Virtual Machine . . . . . 150

    Getting the Best of Both (Quake) Worlds . . . . . 151

    Build Your Own QVM . . . . . 152

Summary . . . . . 154

**CHAPTER 6 CLIENT  
PROGRAMMING . . . . . 155**

Revisiting Weapons: Chain Lightning . . . . . 156

    Trace Your Path . . . . . 162

    Be Like Zeus . . . . . 165

Creating Client Events . . . . . 166

    Enumerating an Event . . . . . 167

    The Communication of `entityState_t` . . . . . 170

    Chain Lightning Lives . . . . . 174

Working with Shaders . . . . . 175

    A Shady Modification: Armor Regeneration . . . . . 177

    Making the Player Glow . . . . . 179

    Regenerating Armor . . . . . 182

Adding New Icons and Sound Effects . . . . .	184
Making It Count . . . . .	185
Getting Up in the Player's Face . . . . .	193
Cache Memory . . . . .	196
Summary . . . . .	200

## CHAPTER 7 DEFEND THE FLAG . . . . 201

The Rules . . . . .	202
What You Will Reuse . . . . .	204
What You Will Create . . . . .	205
Preparing Q3 for DTF . . . . .	205
Your Journey Begins at gametype_t . . . . .	206
itemType_t: Birth of the Sigil . . . . .	209
Fleshing Out the Sigil . . . . .	211
Bending the Rules with powerup_t . . . . .	213
Transforming Flags into Sigils . . . . .	215
Every Item Is a gitem_t . . . . .	215
Sigils Become New Items . . . . .	218
Initializing Sigils for DTF . . . . .	220
Warning Q3 about Sigils (Or Lack Thereof) . . . . .	225
Creating the New Sigil Behavior . . . . .	227
Getting Touchy-Feely with Touch_Item . . . . .	228
Wiring Sigil-Touch for Touch_Item . . . . .	231
Keeping Score with Sigil_Think . . . . .	235
Keeping cgame in Check . . . . .	236
Tricking Q3: Reusing Spawn Points . . . . .	238
The Process of Spawning Level Entities . . . . .	239
Jimmying Item Entities into a Map . . . . .	242
Yanking Out CTF Flags . . . . .	244
Creating a Third Sigil Spawn Point . . . . .	245
Picking the Preferred Item . . . . .	249
Summary . . . . .	252

**CHAPTER 8 WHERE TO GO NEXT . . . 253**

Deathmatch, CTF, and Other Game Types . . . . . 255

- Vanilla Deathmatch . . . . . 255
- Games Without Frontiers: CTF. . . . . 256
- A Class Act: Team Fortress . . . . . 258
- Squad Tactics: Counter-Strike . . . . . 259

Structure of a Mod . . . . . 261

- The PK3 File: Unlocking Its Secrets . . . . . 261
- Art Is Life (and Death, in Q3) . . . . . 264
- Modeling Without a Runway . . . . . 264
- Brushing Up on Level Design . . . . . 266
- Creating Sound Effects Using Household Items. . . . . 268

Summary . . . . . 270

**INDEX . . . . . 271**

## LETTER FROM THE SERIES EDITOR

If you're interested in making 3D games but don't have five to ten years to spend learning 3D graphics, AI, networking, optimization, hardware, DirectX, and about a billion other things, then a great place to start is by creating modifications to existing games. This process is known as *modding*, or developing *mods*.

In a nutshell, there are two engines out there from which 90 percent of all 3D games for the PC are made—id Software's *Quake* engine, and Epic's *Unreal* engine. *Focus on Mod Programming for Quake III Arena* leverages the *Quake III* engine, as it's far more widely used; however, the *Unreal* engine is similar, so the concepts you learn here will translate nicely into any modding venture.

If you've tried to learn modding by reading Internet articles, you've realized that no one who really *knows* how to make mods is writing the articles! The goal of *Focus on Mod Programming for Quake III Arena* is to give you a book written by an expert "modder"—that expert being Shawn Holmes. Shawn has taken a huge subject and condensed it into a nicely-sized data stream that anyone can download into their brain (with proper augmentation of course).

*Focus on Mod Programming for Quake III Arena* starts off by describing what mods are and giving you some history behind the *Quake* technology. Shawn describes key programming concepts so that even non-programmers can get up to speed immediately on creating mods. Then Shawn gets right into it and begins creating weapon mods and using these examples to show you the very complex and mysterious code base of *Quake III*. He picks examples that illustrate concepts such as client server issues, simple physics, game play, and

testing. By the end of the first few chapters you'll be programming in Quake C and will have made numerous mods to the game! Moving on, more complex subjects are broached, such as making modifications to the player, creating new game types, and more.

I can without hesitation say that this is the first book in history that describes in a professional and concise manner how to create mods for the *Quake* engine. Not only do I guarantee that if you read this book, you *will* be able to create mods, but I also believe that you will come away with a deep understanding of how next-generation 3D games are created. This alone is worth the price of admission. Understanding how one of the greatest software engineers in history—that is, John Carmack—thinks is *definitely* a good thing!

So stop reading this and get started on creating a mod for *Quake III*!

Sincerely,

A handwritten signature in black ink that reads "André LaMothe". The signature is fluid and cursive, with a long horizontal stroke at the end of the name.

André LaMothe  
Series Editor



*This page intentionally left blank*

# Introduction

Welcome to the world of mod programming!

The goal of this book is to get you started programming mods for *Quake III*. There is a lot of territory to cover in a short time, but by the time you're finished, you'll have a solid understanding of what it takes to make the necessary changes for your mod. I also plan to give you many examples of how to extrapolate *Quake III* information on your own, so that after you finish this book you can continue to learn by using the methods taught herein. What you will soon discover (as many mod programmers before you have) is that working with *Quake III*'s code is an experiment in learning by itself. I believe that with a bit of dedication, we can clear up the ambiguities and get you rolling your own mods in no time at all.

## How this Book Is Organized

Chapters 1 and 2 of the book introduce mod development, look at some of the best mods already available, and then guide you through the process of setting up your tools. Chapter 3 moves on to some basic programming examples by working with *Quake III*'s weapons, and finally, Chapter 4 introduces you to working with altering player behavior.

Chapters 5 and 6 get a little more advanced. You'll take an in-depth look at the structure of *Quake III*'s code base, and learn how its three major parts communicate. You'll also learn about the Quake Virtual Machine, and its role in mod programming. Then you'll get a chance to work with *Quake III*'s client code in more detail.

Chapter 7 builds on the information in previous chapters, teaching you to develop a custom mod with a completely new game style.

Chapter 8 summarizes elements of the book and goes over what tools you should have under your belt. Breakdown of other mods, as well as summary of external components that go into a mod, is also covered.

If you have some experience developing *Quake III* mods, then you can pretty much read any of the chapters that interest you. If you are new to mod programming, however, there is no better place to start than right at the very beginning of this book.

## System Requirements

If you have played *Quake III* on your machine and it works fine, then your system is probably up to scratch. The required specifications for *Quake III* modding are as follows:

- **Processor.** Pentium II 300+ MHz (Pentium III 500 MHz+ recommended)
- **Operating system.** Windows 95/98/ME/2000/XP
- **Video card.** 8MB RAM (3D graphics card recommended)
- **Memory.** 64MB (128MB recommended)
- **Hard disk.** 500MB minimum for installation of *Quake III* and tools
- **Compiler.** Visual C++ recommended

## On the CD

Included with this book is a CD on which I've compiled some additional resources for your viewing pleasure.

## Source Code

Every chapter in this book that features a tutorial working with the *Q3* source code has its code conveniently saved on the CD, in case your fingers get tired of typing. They are broken out into multiple ZIP files

across separate folders for each chapter. You also have a fresh archive of the complete *Q3* source, which as of this writing is version 1.29h. Additionally, you'll find the latest *Q3* point release on the CD, 1.31.

## **e-Chapters**

*Quake III* Mod programming covers so much territory that I simply could not fit it all into this book! Therefore, I've gone ahead and put some additional chapters on the CD for you. On the CD, you'll find Chapter 9, "UI Programming", which introduces you to working with *Q3*'s User Interface. You'll also find Chapter 10, "Enhancing DTF", which offers some really exciting ways to polish your *Defend the Flag* mod. In it, you'll discover how to create a new HUD to display Flag Statuses, a "Flag Locator", which acts as a compass to guide players towards flags, and build *DTF* support into the *Q3* UI. I've also supplied an Appendix, "Debugging Your Mod in Visual Studio", which offers some expert insight into configuring Visual Studio to program mods more effectively. Appendix B offers a list of online resources for game developers.

## **User-Created Mods**

To give you a feel for the kind of work that is being done in the mod community, I have compiled a set of absolutely insane mods created by fans, just like you. On the CD, you'll find *Painkeep Arena*, *Rocket Arena Q3A*, *Rune Quake*, *Threewave CTF*, and *Urban Terror*.

## **Development Tools**

I've also supplied a few of the major tools that mod developers use, along with their trusty compiler. On the CD you'll find a shareware version of Milkshape 3D that is good to use for 30 days, after which you may register it for a small fee. As well, the latest version of *Q3Radiant* is included, in case you get the itch to tackle a bit of level design.

## Last But Not Least

Thank you for buying this book. I know you'll have as much fun developing your mod as I had putting together all the examples. And if you master everything on these pages, you'll be well on your way to producing a complete mod for *Quake III*—one that is both exciting and fun to play. Remember, anybody can program a modification for *Quake III*, and many of today's game developers started just like you are now. What are we waiting for? Let's go!



# **CHAPTER I**

# **INTRODUCTION TO PROGRAMMING MODS**

**C**reating mods for *Quake III* is a challenging and rewarding experience. It requires a lot of tenacity and hard work, but the payoff is twofold. Not only do you come closer to understanding what goes into professional game development, but you also have the luxury of sharing your programming efforts with hundreds, if not thousands of fans online. In fact, many professional game developers get a head start in the industry simply by playing around with code.

Before you jump in and start developing your mod, let's take a look at what mods are and what drives people to create them, and examine some of the impressive mods that have already been developed. At the end of this chapter, you'll look at the tools you'll need to develop your mod.

## What Is a Mod?

*Mod* is short for *modification*. Simply put, a *mod* is a single change or a package of changes made to a game that alters the way in which the game was designed to work. A mod can involve something as simple as changing the speed at which a rocket moves across a room or as complex as a complete overhaul of the look and feel and even the rules of the game.

Mods came about when fans of the immensely popular id Software game *DooM* started fiddling with the game's code in order to alter how it was played. This feat required a fair amount of dedication considering that *DooM* hadn't really been documented, and that its source code hadn't been made available for download on the Internet. (In fact, the Internet hadn't even become popular yet, and most file transfers were being done over old BBSes, short for *Bulletin Board Systems*.) Even so, dedicated gamers were creating utilities to add new monsters, create new levels, even import new sound effects—ultimately changing how *DooM* was played.

### The Mod Developer Has Arrived

For many of my friends now in the gaming industry, *Quake* was a turning point. These people—boring application developers (or taxi drivers) by day, aspiring mod developers by night—spent countless hours designing levels, coding new bots, and creating special effects—not for financial gain, but to create something that thousands, if not tens or hundreds of thousands of people would play. Many were hired by established development houses on the basis of their mod work, while others went on to form their own game companies. *Quake* really did change things.

id Software saw this as a fantastic way to build onto future game technologies it developed. If fans wanted to work with the company's code, the company would invent better and easier ways of allowing it. Word soon got out during the development of id Software's *Quake* that it would be engineered to be modifiable from the get go. As expected, the release of *Quake* spawned huge amounts of interest among programmers because ID Software allowed the source code to be downloaded and played with. *Quake II* followed in the footsteps of its predecessor, and inevitably so did *Quake III*.

## Why Create a Mod Instead of Just Writing a Game?

So what's with all this mod business, anyway? Doesn't it make more sense to just create your own game from scratch? Not necessarily. For starters, creating a game that is as technically outstanding as *Quake III* is a huge undertaking, requiring considerable amounts of time, money, and manpower. Second, developing a 3D engine takes a lot of coding experience, as well as a deep understanding of math, 3D hardware,



and other associated knowledge. Creating a mod, on the other hand, is a great hands-on way to learn a lot of the skills used in game development. You don't have to create a cool 3D engine; one is already provided.

That means you can focus on creating a game rather than creating the technology. And did I mention it's also a lot more fun?

## NOTE

Throughout the rest of this book, I am going to use the abbreviation *Q3* instead of the long-winded *Quake III*. You never know, those seven extra characters may just make a difference!

## The Tools of the Trade

Before you get started programming your own mods you should know that this book is geared toward programmers with a basic understanding of the C programming language. Don't worry if your C skills are a bit rough around the edges; I'll be sure to go over some more complicated concepts when the time arises.

## Using C

If you've never worked with game source before, you'll be happy to know that I'm going to start from the very beginning, so that you can familiarize yourself with the necessary tools, and play with small bits of code without getting overwhelmed. If you're already familiar with *Q3* and its code, you may want to skip ahead to chapters that focus on the specific area of the code base in which you are interested.

Hopefully, you have at least some knowledge of fundamental C constructs, such as

- **Basic variable types.** Integers, floats, chars, structs, and enums are used quite frequently throughout the *Q3* code base, so it's a good idea to know what these are.
- **Functions.** A segment of self-contained code, a function generally takes input parameters and returns a variable. You should have no problem with the construct of a function, and you

should be able to both identify existing ones and write your own with ease.

- **Pointers.** Quite easily one of the most confusing concepts for beginning programmers, *pointers* are special variables that point to memory addresses. When you see a variable with an asterisk next to it (for example, `*char`), you are dealing with a pointer. Pointers are an efficient way of passing complex variables from function to function, and are used often throughout the *Q3* code base. If you are unclear on pointer usage, I recommend doing a bit of research on the subject.
- **Memory addresses.** Somewhat related to pointers is dealing with memory addresses explicitly via the address-of operator, or ampersand symbol, such as `&char`. If you don't have a good understanding of handling memory addresses, it may be worth it to read up on that as well.
- **Typedef.** This is a keyword in C that allows you to create a new type of variable, which is basically another name for an existing type. *Q3*'s code is designed in such a manner that many variable types are user-defined, so that they better describe their function within the code. In fact, there are quite a few cascading typedefs (a typedef of a typedef of a typedef, and so on) in *Q3*'s code. It's very important to know what is going on when variable types are redefined, so that you can work with the existing types, and go on to create your own when necessary.

If you are comfortable with all these concepts, you're well on your way to programming some modifications in *Q3*.

## Using the VC++ Compiler

To start programming a *Q3* mod, you'll need a compiler. For this book, I have used Microsoft Visual C++ Version 6, because it's the tool of choice for the developers of *Q3* and the majority of mod authors. It is also the easiest to configure. This commercial package is available from most major software retailers; if you are a student at a university or college, you may be able to get it at a discount.

If you have worked with Visual C++ Version 6 (to be referred to hereafter as *VC++*), excellent! Diving into the *Q3* code should be a breeze

for you. If not, no worries: I'll take some time at the beginning of Chapter 2 to help you set up VC++ and get familiar with its interface.

### NOTE

Even though the language I'll use is called Visual C++, don't fret. VC++ is also capable of working with plain-old vanilla C as well. It even has the ability to work with inline assembly language!

## What Mods Are Currently Available?

*Q3* has been around for some time now; consequently, numerous mods are available for it. During the writing of this book I downloaded and play-tested most of them (hard work, I know, but someone has to do it). The following sections briefly review my favorites. I have included these on the CD-ROM that accompanies this book, and recommend that you install and spend a few hours playing them to get some idea of what is possible with the *Q3* engine.

### Urban Terror

On the surface, you could describe *Urban Terror* as simply another *Counter-Strike* clone. Although in some respects this is true, labeling it as such would not do it justice. From the moment you load up *UT*, you realize the developers have succeeded in making it a “total conversion”—that is, a modification that has changed everything in *Q3* including levels, weapons, game types, even the user interface.

*UT*'s attention to detail is spectacular. The current

### NOTE

In case you're not familiar with it, *Counter-Strike* is one of the Internet's most-played mods to date. Although *CS* is a mod for *Half-Life*, the engine that drives it is a derivative of the original *Quake* engine. As an aside, it may interest you to know that Valve Software, the company that created *Half-Life*, hand-picked *Quake* mod programmers to join their company.

version comes with 15 maps, all based on an urban theme, with *UT\_Streets* and *UT\_Docks* being my favorites. The game play revolves around a series of rounds with time limits and different styles of play including Deathmatch, Team Deathmatch, Capture the Flag (CTF), and Follow the Leader. The *UT* mod team plans to add modes of play in the future, and I look forward to its next release. Figure 1.1 shows a player helping his teammates finish off the last of the enemy for a round win.

## Rocket Arena Q3A

*Rocket Arena* goes back a long way. In fact, it was originally a mod created for *Quake*. Then, when *Quake II* was released, *Rocket Arena 2* followed. It only made sense that *Rocket Arena Q3A* would make its debut soon after *Q3* hit the shelves. *Rocket Arena* has always been an exciting mod because it pits you against one other foe, *mano y mano*, in a battle to the death. You start the game by connecting to a *Rocket Arena Q3A* server (or setting one up yourself).



Figure 1.1 A red team member gets the win in Urban Terror.

If others are playing the game, you are placed in a waiting line, or *queue*. While you wait, two other players battle each other until one falls, at which point the loser is removed from the play area and added to the bottom of the queue. The winner remains to battle the next player in the queue; play resumes to see who can survive the longest.

*Rocket Arena Q3A* is very fast paced, and gets your adrenaline pumping. The mod authors added some great features to it, including a built-in MP3 player that enables you to create and organize your own personal music playlists while you frag away!

## Quake III Fortress (Q3F)

*Q3F* is a well-rounded mod that focuses on team play by building on a classic gaming format. Its levels are well thought out, and the unique aspects of the different characters make for a rewarding gaming experience for all styles of players. *Q3F* is a team-based mod that lets you play a variety of character classes, each with its own roles, capabilities, and weapons. These classes include

- **Recon.** This is the fastest unit, but has weaker weapons. The Recon is equipped with a radar scanner.
- **Sniper.** This one-shot killing machine is for the player with quick reflexes; it enables you to take out the opposition from a distance.
- **Soldier.** This is the backbone of every squad—a general all-rounder with some devastating weapons.
- **Paramedic.** No unit is complete without a paramedic. He is reasonably fast, sports some good weaponry, and can heal teammates.
- **Minigunner.** Armed to the teeth with heavy weapons and devastating cluster bombs, this is the mean machine of any *Q3F* team.
- **Flametrooper.** The arsonist of *Q3F*; armed with a flamethrower, napalm launcher, and napalm grenades, he can turn the hardest of foes into crispy pancakes in seconds.
- **Agent.** With the ability to disguise himself as one of the enemy, the agent can slip behind enemy lines and launch a deadly and unexpected attack—making way for the frontal assault.

- **Engineer.** The master of defense, the engineer has the skills to build and maintain auto-sentries that guard the team base and unleash a deadly hail of fire on any approaching enemy (see Figure 1.2). The engineer is also responsible for creating supply depots for the troops.

## Painkeep Arena

*Painkeep Arena* is another extravagant mod that has been around for some time. Like its first incarnation for the original *Quake*, *Painkeep Arena* brings to the table some incredibly detailed and unique level designs, and a quality of game play that is a force to be reckoned with. Some interesting features that *Painkeep Arena* include are as follows:

- **Chain lightning gun.** This is an extremely cool update to the standard lightning gun, causing an arc of lightning to leap from target to target.
- **Turrets.** Much like the auto-sentries found in *Q3F*, these guns target any enemies near them and begin riddling them with bullets.



**Figure 1.2** An auto-sentry guards the corridor in *Q3F*.

- **Bear traps.** As you might have guessed from the name, these are traps you can place on the ground in an effort to trap your enemies, causing damage as they drag their wounded feet around the levels.
- **Gravity well.** An insanely powerful device, this object sucks any and all targets toward it with tremendous speed, causing them to explode on contact in a bloody mess.
- **Airfist.** This unique weapon allows the player to blast concussive forces of air at his opponent and at incoming missiles, sending them in different directions. Airfist can also be used to blast against the ground, propelling the player up to high areas in levels that would otherwise be unreachable.
- **Pork and beans.** The craziest powerup ever in any *Q3* mod, players can pick up and use a can of pork and beans to be restored to full health—with some . . . ahem, side effects.

With its various weapon upgrades and powerups, *Painkeep Arena*, which was put together in a very professional fashion, is a lot of fun. Figure 1.3 shows a player getting busy in one of *Painkeep*'s creative levels.



**Figure 1.3** A player fires the magnum, a new weapon in *Painkeep Arena*.

## Rune Quake

Although *Rune Quake* is a very simple, straightforward mod, it is loads of fun to play. The rules are simple: Play *Q3* as you normally would, on pre-existing levels, with standard *Q3* game rules. There is one slight difference, however. Littered throughout the *Q3* level are *runes*—small objects, usually in the shape of an easily recognizable symbol—that can grant the player some incredible abilities. There are 50 different types of runes, including the following:

- **Rune of fire.** Collect this rune, and you become a walking torch, igniting your opponents with the merest touch—not to mention firing on them with your standard weapons. Water is something you want to stay away from with this powerup.
- **Vampiric rune.** When you hold this rune, you gain health as you damage your opponents, the way a vampire grows strong from the blood of others.
- **Rune of recall.** This rune enables you to set a recall point. Then, when you're in trouble, you can teleport back to your point. It's great for emergency escapes!
- **Switch rune.** This sneaky rune, when activated against an opponent, enables you to switch with them—locations, weapons, health, even powerups!
- **Weirdness rune.** Keep your distance from any player dropping colored bubbles—a sure indication he holds the Weirdness rune. If you get caught in the bubbles, you can expect to bounce, wobble, weave, fly, spin, and just plain get sick!
- **Rune of uncontrollable jumping.** If you hold this rune, any players near you start hopping against his will, which is sure to throw his aim off.
- **Phase rune.** One of my favorite runes in *Rune Quake*, the Phase rune allows you to pass through thin walls and surprise your opponents (see Figure 1.4). Note the transparent-weapon effect, which indicates that the player has become a ghost and can shift through solid surfaces. The rune's icon, situated in the lower-right area of the screen, mirrors that effect.

This is just a smattering of mod examples that regular fans of *Q3* (like you) have gone on to create. I've put these mods on the CD-ROM so you can try them out yourself, and I highly recommend you scour the





**Figure 1.4** A player makes a surprise kill after phase-shifting through a wall in Rune Quake.

Internet for even more. In Appendix A on the CD-ROM, I'll provide you with some online resources to help you locate other mods so you can get a feel for what's out there.

## Summary

Game programming is both exhilarating and challenging, and the best experience you can get is starting with mod development. The best way to prepare yourself for modding (both physically and mentally) is to take a look at what kinds of mods have already been created. This can often get you thinking about “what is fun” and “what can be done better.” It also saves you from remaking a mod that is already out there. Players are constantly looking for new and exciting ideas to be implemented in mods, so don't be afraid to explore uncharted waters. So, without further ado . . . let's get making some mods!



# CHAPTER 2

# C PROGRAMMING IN QUAKE III

**A**lmost every professional game written today is in C or its big brother, C++. C is used for many reasons. First, it's fast. Short of writing your code directly in assembly language, squeezing the most performance out of your programs will definitely be easier when written in C. When you consider that id Software, the creator of *Quake III*, is well-known for cranking out code that taxes the latest and greatest computer hardware to its very limits, it's no wonder that C is chosen for its speed. Second, it's standardized. That means there are compilers for C on all kinds of different operating systems. Code that is written in C for one platform can be translated, or ported, to another platform with great ease. *Q3* also benefits from this because there are multiple ports available for users on operating systems other than Win32, such as Linux and MacOS (for Apple Macintosh).

Learning C is also essential if you think you might later decide to move on to C++, because C++ is actually an extension of the C language. As I write this, the programmers at id Software are pushing the envelope with their next big project, an updated version of *DooM*, and the word on the street is that it will be written in C++. That, coupled with the fact that the next *DooM* will almost certainly be just as modifiable (perhaps more so) as *Q3*, should inspire you to get in on the C++ action.

## NOTE

When I use the term Win32, I mean any 32-bit Windows operating system produced by Microsoft. In layman's terms, this is any operating system produced since Windows 95, including Windows 98, Windows Millennium, Windows NT 4.0, Windows 2000, and Windows XP.

# The History of Quake and Its Code

John Carmack, lead programmer at id Software, is the man responsible for creating the technology that drives all the latest and greatest games. Not only do his 3D engines power id Software's games, such as the *Quake* series, it also powers many other companies' games as well, thanks to licensing agreements made between those companies and ID Software.

Carmack was determined from the onset of development for the original *Quake* that the source would be readily available to mod authors, and easily used. You may be wondering, however, how id Software managed to make money on *Quake* given that the game's source code was released. After all, couldn't anybody just download the source and make his own games? The answer is simple: When *Quake* was designed, it was built as two separate interlocking components: the 3D engine and the game logic. The code that drove the 3D engine remained proprietary, and was locked away within the dark confines of id's office space. (Well, id Software's games are spooky, so why wouldn't its office space be?) Meanwhile, the code responsible for monsters' behaviors, how the weapons worked, and the rules of the game was made available for public consumption.

John Carmack had an extremely innovative idea for implementing the code for *Quake*—he created a simpler version of C that would be used to code the logic for *Quake*, complete with its very own compiler. This language, called "QuakeC," was then used by mod authors to create mods for *Quake*. The process involved getting the current code from id's public FTP site (which is still available today), along with the QuakeC compiler, called "qcc." Each part

## NOTE

Carmack has always felt that anybody who wants to learn from other people's code should be able to. For this reason, he has always made it a rule to release all the source (including the 3D engine code) to the games he's built once enough time has passed. As of this writing, he has not only made the full source to *Quake* available, but *Quake II* as well!

of *Quake's* game logic was contained in different qc files, such as world.qc and client.qc, which consisted of various bits of code written in John's custom language. Then, when various qc files had been successfully modified, they were compiled down into a progs.dat file, which was then placed in *Quake's* install directory within its own folder, such as C:\quake\mymod\progs.dat. Finally, *Quake* would be fired up with the `-game mymod` parameter, where mymod was the name of the new folder, and voilà! The newly modified code would come to life within *Quake*.

## The Move from DOS to Win32

After *Quake* was released, it went through many evolutions. For instance, thanks to the development of a programming standard for 3D objects called OpenGL, *Quake's* 3D engine went from being software accelerated (meaning the CPU was solely responsible for drawing the world) to hardware accelerated (meaning the video card contained special processors that rendered the world faster and more efficiently). It also evolved from a basic dumb client-server model to a model in which the client attempted to predict events, thereby reducing latency or *lag* while being played online with a 56Kbps modem. Finally, it got a well-deserved boot up from the dark ages of DOS to a full Win32 application, called WinQuake. After playing with the options that Win32 gave them, the programmers at id decided they would move to Win32 for the development of *Quake's* sequel, aptly titled *Quake II*.

The move to Win32, however, meant the demise of *QuakeC*, because Win32 required using Visual C++—and it didn't seem to understand *QuakeC*. As a result, a new question arose: How could the game logic be most efficiently separated from the 3D engine so that mod authors could use it? The answer would be the DLL, or dynamically linked library.

### NOTE

OpenGL, developed by Silicon Graphics, Inc., is an Application Programming Interface, or API. Because it was an open standard, OpenGL was regarded favorably by the programmers at id—they wanted to be able to port *Quake* to other platforms. By using a standard graphics library, that feat would be less ominous. John Carmack still uses OpenGL today.

In Win32 C programs, parts of code that a programmer deems reusable can be wrapped up in a separate file, which makes itself available via hooks called imports and exports. This separate file is called a DLL, and ends with the .dll extension. When another programmer wants to use that code, he can create a program that hooks into the code during runtime and use it as if it were actually written into the main program. This is called *dynamically linking*. These DLLs can then be packaged with an archive utility and uploaded to the Internet for others to share; alternatively, if the author of the DLL deems its value high enough, it can be sold. Because id needed *Q2*'s proprietary 3D engine to remain a secret, the logical choice was to allow the game code to reside in a DLL.

When *Q2*'s game source was made available, mod authors snatched it from the same FTP site that hosted the original *Quake* source. It was structured in very much the same manner as the old QuakeC code was. The code that took care of specific components within *Q2*, such as weapons, monster control, and game rules, were broken up into standard C files, ending in the .c extension (such as `g_weapon.c` and `g_monsters.c`, in which the letter `g` represented the word game).

After various files were modified and were ready to be tested, they were compiled into a DLL called `gamex86.dll` (x86 represents the architecture of the CPU—in this case, the Intel x86 processor found in common PCs). The `gamex86.dll` was then placed in its own directory within *Q2*'s install directory, as in `C:\quake2\mymod\gamex86.dll`. *Q2* was then fired up with the `+set game` parameter, with the value of the game parameter equal to the new directory, like so:

```
quake2.exe +set game mymod
```

## NOTE

Breaking out various chunks of code into reusable DLLs was such a good idea, the programmers at id did the same thing to allow multiple 3D rendering drivers for the various types of hardware that were popular during *Q2*'s reign. One was a software driver (`ref_soft.dll`) that made *Q2* render or “draw” its world like the original *Quake*: completely CPU based. The other was `ref_gl.dll`, the OpenGL hardware-accelerated driver. Other vendors provided drivers to be used with *Q2* in the same manner, such as *Vérité* and *PowerVR*.

This code would trigger the loading of the new DLL as *Q2* came to life on-screen.

## Hello Quake III, Goodbye DLL! (Sort Of)

The move to Win32 was a success, so id Software decided to keep the momentum going. Shortly after *Q2* was released, the company announced that work had begun on a third installment in the *Quake* series, also to be written in Win32 C. John Carmack, however, was not entirely convinced that using a DLL was the right architectural choice, the main reason being security. Because a DLL, by nature, makes itself available to other programs, it is capable of calling other DLLs. Any kind of functionality can be written into a Win32 DLL, from the capability to create a game on your desktop, to defragmenting your hard drive, to creating an email client, and so on. That means that malicious users could potentially write evil code into a DLL, link it to the DLL used in *Q2*, and cause some pretty nasty results. Who's to say someone couldn't write a mod that secretly included a call within its DLL to format your hard drive as soon as you scored 100 frags in a game of deathmatch? Not a pretty picture, to say the least.

So John went back to the drawing board to see what he could come up with for *Quake III Arena*. His solution was quite innovative. He proposed that *Quake III Arena* run a Quake Virtual Machine (QVM) within the core of the engine. This QVM could then interpret a special kind of compiled code within the game, aptly named QVM files. He went on to say that a QVM would be totally safe, as they would only be interpretable from within *Q3*. No external Win32 functions could be called, such as formatting a hard drive. He also explained that a QVM file would be made from exactly the same code that could compile into a Win32 DLL, and he would provide a custom compiler to create this QVM. As a result, mod authors could not only write mods in their familiar C, but they could compile into both a Win32 DLL and a *Q3*-specific QVM. The QVM, then, would be the final product that mod authors would upload to the Internet to share with the gaming community.

And this is where you come in. . . .

**NOTE**

*Quake* not only said goodbye to the DLL in its third installment, but one other concept went away with the release of *Q3*: It is the first *Quake* game in the series that has no support for a software renderer. *Q3* contains only 100-percent hardware-accelerated 3D graphics.

## Getting Set Up

Now that you know a bit of *Quake*'s history and how its code has evolved over time, let's take a look at the code itself. You'll start by getting your compiler configured. Your compiler will be your best friend for the remainder of this book, so ensuring that it is set up efficiently is key. *Q3* is very particular about where and how things should be installed (and uninstalled), so please read the next few sections carefully.

**TIP**

Before you start, I suggest you visit this book's Internet site (<http://www.moddeveloper.com/downloads/>) and download the *Q3* source; unfortunately, due to id Software's End User Licensing Agreement (EULA), I cannot include it on the CD-ROM. While you are there, take a look at the updates section (<http://www.moddeveloper.com/updates/>) to see if anything has changed. id Software occasionally releases a source update that alters the directory names or lacks certain files needed for mod development.

## Installing *Q3* and the Source

To get set up, you must first install the game in the correct location, add the point releases, and finally add the *Q3* source. By default, *Q3* installs itself in C:\Program Files\Quake III Arena\. Unfortunately, when you are developing mods, they need to be in a directory named "quake3" in the root of your hard drive. This can be any drive—for



example, C:\quake3, D:\quake3, and so on—but the folder must be named “quake3.”

If you have already installed *Q3* to its default location, you’ll need to uninstall it and then re-install it in a quake3 folder in your root drive, as noted in the preceding paragraph. When uninstalling *Q3*, however, it is extremely important that you remove the individual elements in the order listed here (failure to follow these steps in order can result in elements being uninstalled incorrectly and an unstable environment if *Q3* is reinstalled):

1. If you have installed any of the game source files such as *Q3AGameSource.exe* or *Q3A\_TA\_GameSource\_129h.exe*, remove them using Add/Remove Programs in Control Panel.
2. Remove any installed point releases. These will be labeled *Quake III Arena Point Release (1.31)* or something similar.
3. Remove *Quake III Team Arena* if installed.
4. Remove *Quake III Arena*.

When you are ready to install the source, simply double-click the source’s executable file (the latest one being *Q3A\_TA\_GameSource\_129h.exe*) and follow the dialogs it presents to you. This will involve:

1. Clicking the Next button on the first Welcome dialog.
2. Clicking the “Yes” button after reading and agreeing to id Software’s EULA.
3. Selecting the proper *Q3* installation directory (which, by default, should be C:\quake3\ ) and clicking Next.
4. Clicking Next a final time to confirm the installation of the source, finishing by clicking Close to close the dialog when it is complete.

When you are certain your copy of *Q3* is in the correct location and that it is working properly, you will need to install the latest patch, Point Release 1.31. This patch can be found on this book’s CD in the patches directory. After the point release is installed, you can add the *Q3* source. Run the installer and select the quake3 directory where the game is located.

To successfully build the *Q3* source and deploy Quake Virtual Machines (more on these later), you will need to install a few additional files. I have compiled these into a single ZIP file—*buildupdates.zip*, found in the `\files\patches\` directory—that you can extract to your *quake3* folder. Do this now (make sure that you select Use Folder Names in your ZIP program). This will install:

- `cpp.exe` and `rcc.exe`, two files used during QVM compilation (which you'll cover in Chapter 5).
- `game.bat`, `cgame.bat`, `q3_ui.bat` and `all.bat`, four batch files to assist you in QVM compilation.
- Various “Project” files that will compose the *Q3* source's workspace in Visual Studio.

With *Q3*, the point release, and the source installed, you are almost ready to load your compiler and build a mod. But first, let's look at the components that make up the *Q3* source and prepare a working environment.

## The Source Directory

In your *quake3* folder, you will find a directory called `code`. This contains the entire source to build *Q3* and *Q3 Team Arena*. Make a copy of this folder and name it `MyMod`; you will use this as your base for development throughout this book.

### NOTE

By copying the folder rather than simply renaming it, you ensure that if you delete a file by accident or wish to change something back to the way it was, you have a clean copy of the original.

### NOTE

Chapter 5, “Quake Communication,” goes into more detail about the Quake Virtual Machine and how to use the batch files to create QVM files.

The `MyMod` directory contains a series of files and additional directories, listed in Table 2.1.

**Table 2.1 The Q3 Source Files**

<b>Name</b>	<b>Description</b>
game/	This directory contains the source for the Q3 server.
cgame/	This directory contains the source for the Q3 client.
q3_ui/	This directory contains the user-interface (UI) code for <i>Quake III Arena</i> .
ui/	This directory contains additional UI code, used for the Team Arena Expansion Pack.
Project.dsw	This project workspace file is created for Visual Studio. It loads all the source code for Q3 and is setup for compiling your mod.
game.bat	This batch file builds the Quake Virtual Machine for the server.
cgame.bat	This batch file builds the Quake Virtual Machine for the client.
ui.bat	This batch file builds the Quake Virtual Machine for the UI.
all.bat	This batch file builds all the QVM files (game, cgame, and UI).

## Using Visual C++

Because you'll use VC++ to build mods throughout this book, let's focus on its installation and setup here. Do the following:

1. Insert your installation CD into a CD drive.
2. Run the installation CD's setup file, typically named "setup.exe" and found on the root of the disc. You should see a Welcome installation screen; click Next to continue.

### NOTE

Microsoft offers VC++ on multiple distributions; this section details the installation of the version I use, Visual Studio Enterprise. Your installation screens may vary slightly from the screenshots I've included, but they should all contain similar options that will lead to the same end result.

3. Visual Studio's EULA screen should prompt you to accept the terms and conditions of installing the software. Go ahead and click the "I accept the agreement" radio button and click Next.

Enter the required name and registration information for VC++, and click "Next."

4. If given the opportunity to choose what type of installation you want, select Custom; this enables you to specify which pieces of software are installed. Click Next to move on.
5. You are asked to specify the folder in which you want to place the software's common files. Feel free to accept the default path—in this case, C:\Program Files\Microsoft Visual Studio\Common. Click Next to continue.
6. Some distributions of VC++, including the Visual Studio Enterprise distribution, are packaged with other software; your focus, however, is VC++. To install only the software you'll need, uncheck everything but the Microsoft Visual C++ 6.0, Enterprise Tools, and Tools check boxes, as shown in Figure 2.1. Click Continue when you've picked the noted software.
7. If given the opportunity, click the Register Environment Variables check box to select it, as shown in Figure 2.2. This is to allow Visual Studio to "find" files across various folders on your system that it requires to compile code. Click OK to continue.

**NOTE**

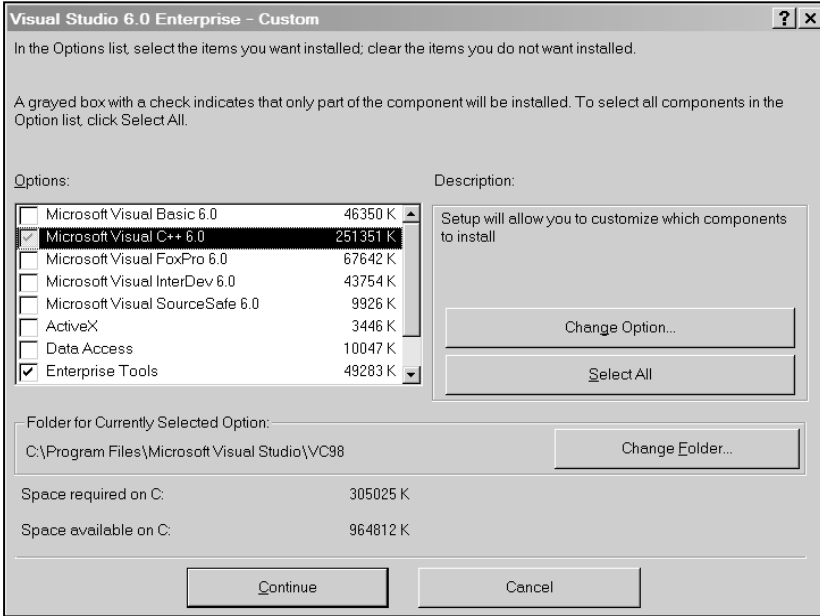
If your CD drive's auto-run feature is enabled, you may not need to run the setup file yourself; instead, it will be started automatically.

**NOTE**

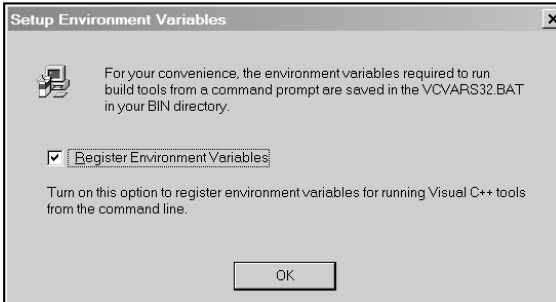
The screen shown in Figure 2.1 also enables you to change the installation folder of VC++ itself, but you can leave the default choice as is.

8. The VC++ files are installed on your hard drive. When this operation is complete, you may be informed that an icon has been installed on your start menu (within the same program group as VC++) that will allow you to set up debug symbols. You can click OK to continue.

## 24 2. C Programming in Quake III



**Figure 2.1** Umm ... I'll just stick with Visual C++, thanks.



**Figure 2.2** Yes, I will register my environment variables, thank you very much.

**TIP**

Debug symbols can often help you determine how or why a program fails, so be sure to run that setup after you are finished with your VC++ installation.

9. If prompted to reboot, go ahead and do so.
10. The installation may prompt you to install MSDN, which is a good idea as it has an extensive repository of programming information that can be referred to when in need. Any other options that are offered can be skipped. Go ahead and continue to click Next until you get to the Finish button, which should conclude the entire installation.

Although you are officially finished installing VC++, I highly recommend checking the Internet for any service packs. Just as id Software releases patches for *Q3*, so too does Microsoft release updates for its compilers. As of this writing, the current Service Pack available for VC++ is SP5 (be sure to select the “Full” one), and you can download it from <http://msdn.microsoft.com/vstudio/downloads/updates/sp/vs6/sp5/default.asp>. You have the option of downloading multiple separate EXEs that will combine during the patch, or of downloading one 126MB file. Either selection is fine, although the latter can be quite intimidating—especially if you don’t have a high-speed connection to the Internet.

After the file is downloaded and extracted to a temporary directory, run `setupsp5.exe` to install the Service Pack. Fortunately, running this setup procedure isn’t nearly the hassle that running the original VC++ installation is. It should do everything in one fell swoop. When this setup routine is complete, you are ready to tackle *Q3* programming.

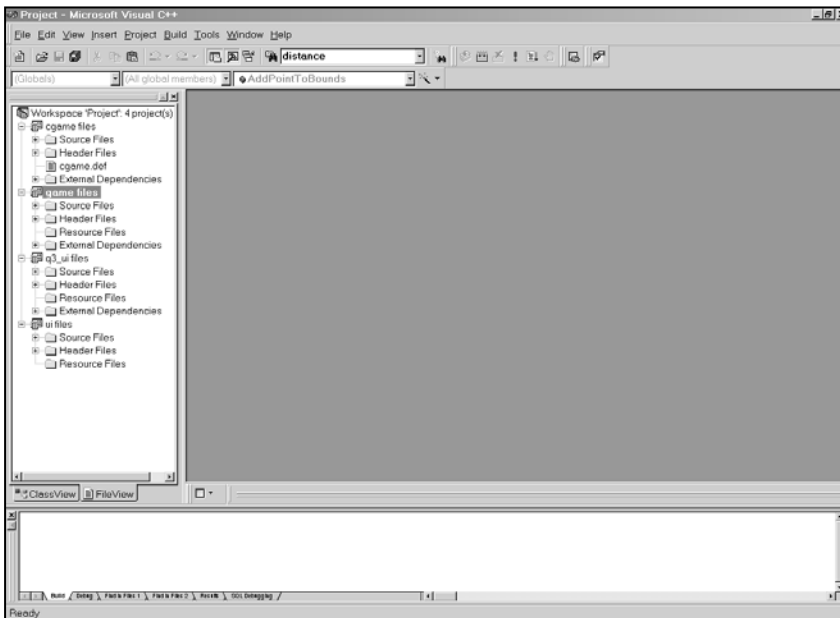
## Building the Source

By now, you should have VC++ installed and fully patched, raring to go. As an added bonus, the project workspace provided with this book has been completely configured so that you don’t need to mess

around with any of the complicated settings; just follow the next set of steps to build the source.

Let's get into the meat of mod programming by making a quick mod to *Q3*'s rocket launcher.

1. Double-click on the Project.dsw file in your /MyMod/code/ directory. Visual Studio is loaded for you.
2. The Visual Studio Integrated Development Environment (IDE) appears. In the top-left corner of the screen, click on the File View tab under the project.
3. As shown in Figure 2.3, a folder list appears containing four separate directories, or projects, which are used to build separate components for *Q3*. Each project includes all the source files for the client, server, and both user-interface modules, labeled cgame files, game files, q3\_ui files, and ui files, respectively. Because you are primarily interested in making changes to the server-side code, double click on the game files project.



**Figure 2.3** The newly opened workspace in Visual Studio, with game files as the active project

4. The game files project displays three folders, labeled Source Files, Header Files, and Resource Files. Double-click on the Source Files folder. A list of the individual source files that control every aspect of the *Q3* game will appear.
5. To make a mod of the rocket launcher, scroll down to the `g_missile.c` entry and double click on it to open the file. A new window will open containing the source to the projectile weapons. This controls how they fire, the damage they do, and the effects displayed on-screen.
6. Scroll down to line 646, where you'll see the following:

```
bolt->s.pos.trType = TR_LINEAR;
bolt->s.pos.trTime = level.time - MISSILE_PRESTEP_TIME; // move a
bit on the very first frame
VectorCopy( start, bolt->s.pos.trBase );
VectorScale( dir, 900, bolt->s.pos.trDelta );
SnapVector( bolt->s.pos.trDelta ); // save net bandwidth
VectorCopy( start, bolt->r.currentOrigin);
```

This code sets the speed at which the rocket moves across the room when fired from the rocket launcher. The speed is set to 900; change it to 200, as shown here:

```
VectorScale( dir, 200, bolt->s.pos.trDelta );
```

7. Save the file by clicking on the Save button (the one with a disk on it) on the toolbar at the top of the screen. Alternatively, open the File menu and choose Save, or press Ctrl+S.

Congratulations! You have just made your first *Q3* mod. All you need to do now is compile the changes and load them into *Q3*.

## Compiling the Project

Because there are several projects loaded into the workspace, you have a number of options. You can compile all four projects at once, which would produce three new DLLs for you (only one UI DLL is generated, based on the type of project you are working on). Alternatively, because you have modified a file in only one of the projects, the game project, you can choose to compile only that project. To do so, do the following:

1. Open the Project menu, select Set Active Project, and click game.



2. Specify whether you are building a release version of the code as opposed to a debug version by opening the Build menu, choosing Set Active Configuration, and choosing game-Win32 Release.
3. Open the Build menu and choose Rebuild All to compile the mod. The compiler compiles all the files into a DLL file; the progress of this operation is shown in an output window at the bottom of your screen.

When the compile operation is finished, you should see the following message:

```
qagamex86.dll - 0 error(s), 0 warning(s)
```

This means that the source has been built and that no errors were found in the code. If you look in your MyMod directory, you will also find a new folder named Release, in which you should find a brand new qagamex86.dll.

## NOTE

If you chose to install the Q3 source to a folder outside the game installation directory (as I often do), you will be responsible for manually copying your created DLLs over to your MyMod directory. With the game-Win32 Release build used in this example, you should find qagamex86.dll in a folder named Release within the code folder where you installed your source.

## Loading Up Your Mod

The time has come to test your creation. Bring up a command prompt and change directories to your quake3 folder. You can now load up your mod by typing the following:

```
quake3.exe +set fs_game MyMod +map q3dm1
```

Once in the game, pick up the rocket launcher and fire away. You should see that the rockets are moving noticeably more slowly.

**TIP**

**A fast way to run your changes is to use a batch file. Simply right-click an unused area of your desktop and choose New Text Document from the shortcut menu that appears. Name the document MyMod.bat. Then, right click it, select Edit, and type `quake3.exe +set fs_game MyMod +map q3dm1`. Save your changes and close the document. You now have an instant way to load your changed code into Q3.**

## Looking at the Quake III Code

How addictive is that? Come on, admit it. You just spent the last ten minutes running around, firing rockets, and standing in front of them, didn't you? Well, while you are waiting for the men in white coats to come knocking on the door, let's take some time now to look at the *Q3* code in greater detail.

One of the hardest things about developing a mod is knowing where to start. As your modding journey continues, I will provide some general suggestions on how to plan your development. To make sure things don't get too serious, you'll finish up by building your very own homing-missile mod.

### On Your Marks, Get Set . . .

*Q3* has about 150 files that control every aspect of the game. Unfortunately, making a change or adding a new weapon can often mean adding code to dozens of them. This can seem at first like quite a daunting task, but don't be put off—after a while, some of the changes you make will seem like second nature.

When you build a homing missile later in this chapter, you will be introduced to some important concepts such as flags, entities, and the fact that weapons “think.” Before you get to that, however, I suggest you spend an hour or so with the next section and browse through

the individual files. Don't worry—there may be some things you don't understand, but you will pick up valuable information on how *Q3* works.

## The Q3 Source

Listing all the *Q3* files and the functions they include could easily fill a book this size. Instead, I have picked the most important ones and briefly summarized each one. Table 2.2 lists the game project files; these make up the core of any *Q3* mod.

**Table 2.2** game Files

---

<code>ai_cmd.c</code>	This file controls how the bots react to commands issued by teammates and other players as well as functions for issuing the actual commands.
<code>ai_dmnet.c</code>	This is the core of how bots achieve their goals. Take a look at the <code>BotClearPath</code> function on line 1306; it shows how the bots go about clearing obstacles such as proximity mines.
<code>ai_dmq3.c</code>	This file contains more great bot code for goal-orientated behavior such as playing Capture the Flag. It also includes a variety of tests and functions including weapon selection, inventory, item control, and general movement decisions.
<code>ai_main.c</code>	This file contains bot initialization code.
<code>ai_team.c</code>	This is the starting point for the team code. It is responsible for issuing orders, assigning tasks, and so on.
<code>bg_pmove.c</code>	This file handles player movement and physics routines. It controls how the player behaves when running, walking, swimming, sliding, and so on.
<code>g_active.c</code>	This contains the client think events. This is where any player events are processed, such as taking damage, using items, or activating triggers.
<code>g_client.c</code>	This file functions to spawn the client and handle the connection with the server.

<code>g_cmds.c</code>	This file controls the sending of messages to the client, such as adding it to a team, giving an item, and calling for a vote.
<code>g_combat.c</code>	This file handles client combat, death, and damage. Have a look at the function <code>LookAtKiller</code> , which is run when the player dies so you can see who the attacker was.
<code>g_items.c</code>	This file specifies how items such as weapons, ammo, and powerups are picked up, dropped, and respawned in the game.
<code>g_main.c</code>	This file handles general game control such as loading maps, adding players to a tournament, and voting. Have a good read of this to get an overall feeling for the structure of <i>Q3</i> .
<code>g_missile.c</code>	This file functions to control the projectile-based weapons. You will be using this file in the homing missile example.
<code>g_mover.c</code>	This file handles entity movement routines, doors, triggers, and platforms.
<code>g_session.c</code>	This file stores and retrieves information that needs to be persistent across multiple level loads.
<code>g_spawn.c</code>	This file handles spawning entities.
<code>g_svcmds.c</code>	This file handles commands that can be executed by the server console but not remote clients, such as IP filters and bans, forcing players to a team, and adding bots.
<code>g_team.c</code>	This file handles general team functions such as adding scores, getting a flag's status, and so on.
<code>g_trigger.c</code>	This file handles trigger functions including the teleport and timed triggers.
<code>g_utils.c</code>	This is a collection of utilities used by many of the other files, including the very important <code>G_Spawn</code> function, which handles adding new entities to the game world.
<code>g_weapon.c</code>	This is a collection of events that happen when a weapon fires, such as dealing damage, or creating specific effects.

Table 2.3 lists the `cgame` files, which handle the client-side operations in *Q3*. I will get into a much more thorough discussion regarding `cgame` files in Chapter 6, “Client Programming;” for now, just browse through these files.

**NOTE**

Whenever I refer to *gibs*, I mean the shower of fragmented body parts that spray the screen after a player explodes. Sounds yucky, but it has been a term used by players since as far back as *DooM*. It's pronounced with a j sound, as in the word *giant*.

**Table 2.3** `cgame` Files

<code>cg_consolecmds.c</code>	This file handles local console functions for processing key binds or typed-in commands.
<code>cg_draw.c</code>	This file handles local drawing commands for displaying HUD items, scoreboards, and so on.
<code>cg_effects.c</code>	This is used to generate local effects such as bubbles when entering water, particles, smoke, blood, gibs, and explosions.
<code>cg_ents.c</code>	This file handles local entity functions that are processed every frame.
<code>cg_info.c</code>	This file functions to display the loading screen between levels. It shows icons, status, and so on.
<code>cg_localents.c</code>	This file generates the commands for the renderer to display entities such as smoke, gibs, and shells.
<code>cg_main.c</code>	This is the entry point for the client-side game. It loads the graphics, shaders, models, menus, and sounds.
<code>cg_players.c</code>	This file handles the animation and sounds for the player.
<code>cg_predict.c</code>	An important element of a multi-player game is the conservation of bandwidth. The <i>Q3</i> server, instead of updating the client with every move an entity makes, sends a snapshot every so often. The functions in this file are responsible for filling in the missing information; this is called <i>interpolation</i> .
<code>cg_scoreboard.c</code>	This file displays the scoreboard on top of the screen.
<code>cg_weapons.c</code>	This file is responsible for local weapon handling such as selecting the next weapon, drawing trails, and ejecting shells.

**NOTE**

I have not covered the user interface files here, because they are pretty self-explanatory. Their function is to display menus and the heads-up display or HUD to the players. In Chapter 9, “UI Programming,” you’ll build a custom interface for your mod; I’ll go into more detail then.

## Planning

The key to developing any mod—be it a simple weapons enhancement or a full-blown total conversion—is research and planning. I have often started developing a function to provide a certain feature only to discover that the folks at id have already written it, even if it is being used for a different purpose. When planning out your mod, try to identify the separate elements you will need to make it function, then look through the source to see if anything similar exists. Describing its purpose exactly often helps with this process.

## A Simple Mod: The Homing Missile

In the rest of this chapter, you’ll focus on building a homing-missile mod. By having you do this I aim to show you the approach I take when developing mods, and to introduce you to a few fundamental features of *Q3*.

Because you will be creating a homing missile, it is probably wise to figure out how the standard *Q3* missile functions. First, however, you might need a refresher on just what a function is and what it does. A *function* is a block of instructions that can be called from another point in a program, and is used to structure a program in an organized fashion. Commonly used algorithms can be wrapped into a function, which can be called from different parts of a program, saving the programmer from having to duplicate code.

The format of a function is as follows:

```
type name (argument1, argument2, ...) { statement }
```

In this format, type is the type of data returned by the function, name is the name of the function, arguments are parameters passed to the function, and statement is the body of the function. Here is an example of a function:

```
int add (int a, int b)
{
    int res;           // a variable to store the result
    res = a + b;      // add a and b and store in res
    return res;       // return the result
}
```

This code defines a function called add; this function accepts two parameters, a and b, which are integers. The body of a function is contained within curly braces; in the example above, a new variable called res is defined, a and b are added together and stored in res, and res is then returned to the code that called the function.

The function could be called from another part of the program using the following:

```
int r;                // A variable to store the result
r = add(5,18);        // Call the function
```

In this function, 5 and 18 are passed to the add function. A result of 23 is returned and stored in r.

Now that you understand functions, open up g\_missile.c in the game project, and go to line 621. You'll see a function called fire\_rocket; it's responsible for creating an entity and defining it as a rocket (also called a missile). This function is shown in the following code snippet; I have annotated the code so you can understand what it does.

```
1. gentity_t *fire_rocket (gentity_t *self, vec3_t start, vec3_t dir)
{
2.     gentity_t *bolt;
```

## NOTE

Functions in Q3 tend to be a lot more complicated than this, often passing whole objects rather just variables, but the principle is the same.

```
3.     VectorNormalize (dir);
4.     bolt = G_Spawn();
5.     bolt->classname = "rocket";
6.     bolt->nextthink = level.time + 15000;
7.     bolt->think = G_ExplodeMissile;
8.     bolt->seType = ET_MISSILE;
9.     bolt->r.svFlags = SVF_USE_CURRENT_ORIGIN;
10.    bolt->s.weapon = WP_ROCKET_LAUNCHER;
11.    bolt->r.ownerNum = self->s.number;
12.    bolt->parent = self;
13.    bolt->damage = 100;
14.    bolt->splashDamage = 100;
15.    bolt->splashRadius = 120;
16.    bolt->methodOfDeath = MOD_ROCKET;
17.    bolt->splashMethodOfDeath = MOD_ROCKET_SPLASH;
18.    bolt->clipmask = MASK_SHOT;
19.    bolt->target_ent = NULL;
20.    bolt->s.pos.trType = TR_LINEAR;
21.    bolt->s.pos.trTime = level.time - MISSILE_PRESTEP_TIME;
22.    VectorCopy( start, bolt->s.pos.trBase );
23.    VectorScale( dir, 900, bolt->s.pos.trDelta );
24.    SnapVector( bolt->s.pos.trDelta );           // save net bandwidth
25.    VectorCopy (start, bolt->r.currentOrigin);
26.    return bolt;
27. }
```

Lines 1 and 2 are the entry point for the function; it expects an entity to be returned, and is passed a starting location and a direction to face. Line 3 “normalizes” the vector by making its distance exactly 1.0 (we will cover vectors in more detail in Chapter 3). Lines 4 and 5 call a function that either finds a free entity or creates one, and defines it as a rocket. Lines 6 and 7 set the time until the next think period (I’ll get to that in a minute), and specifies which think function will be called.

**TIP**

Although floating points (also called floats) and integers are both numeric variable types, their main difference is that floats can contain a decimal place, whereas integers are constrained to whole numbers only.



Lines 8–12 define the type of bolt, link position, the weapon it was fired from, and its owner. The amount of damage is then set in lines 13–19, with the indirect damage that occurs if the rocket explodes nearby (referred to as *splash damage*). Finally, in lines 20–25, the rocket is moved forward slightly, then fired at 900 units/second.

You just saw how spawning a new entity and setting its parameters created a rocket. This is just part of the process. If you open `g_weapon.c` and scroll down to approximately line 372, you will find a function called

`Weapon_RocketLauncher_Fire`. As you might have guessed, this has something to do with firing the rocket. This function appears in the next bit of code.

```
void Weapon_RocketLauncher_Fire (gentity_t *ent) {
    gentity_t      *m;

    m = fire_rocket (ent, muzzle, forward);
    m->damage *= s_quadFactor;
    m->splashDamage *= s_quadFactor;
}
```

The code in this listing is called when the player has the rocket launcher in his hand and presses fire. It calls the `fire_rocket` function listed previously, and multiplies the damage by any powerups the player has.

There is one last function you can look at: `G_ExplodeMissile`. It's found in `g_missile.c` starting at line 41. This is the missile's think routine, which is called after 15 seconds of not hitting anything. It destroys the missile and frees up the entity so that there are not hundreds of them flying through space. So, what is the deal with this "think" stuff, anyway?

## I Think, Therefore I NextThink . . .

Any object, created or spawned and placed within *Q3*, that has some kind of behavior associated with it is said to *think*. This think variable is often set when an entity is first created. In reality, the think variable actually points to a function that the particular object will run, at vari-

### NOTE

Time in *Q3* is measured in thousandths of a second. 1000 would equal 1 second; 15000 is, therefore, 15 seconds.

ous times during execution of the game. So, different types of objects naturally have their `think` variable pointing to different functions. For example, in the `fire_rocket` function you looked at, the rocket's `think` variable pointed to a function called `G_ExplodeMissile`, which basically tells `Q3`, "When this rocket's time is up, call the `G_ExplodeMissile` function." The next question, then, is when is the rocket's time up?

Just as the `think` variable is set when an entity is spawned, so too is the interval in which it needs to call the function that `think` points to. Some entities behave in such a manner that they need to have their `think` function called constantly, such as 10 times per second. Others, like the rocket in the first bit of code listed above, need to update `think` only once every 15 seconds. The size of this interval is set with the `nextthink` variable, which is really an integer representing a value of time (in milliseconds) in the game. In the rocket's code above, the `nextthink` is set to the value of another variable called `level.time`, plus an additional 15000 milliseconds. What this says to `Q3` is, "The instant this rocket is created, I want it to think exactly 15 seconds from now." As you might guess, `level.time` is a global variable that holds the current time in milliseconds.

So, by taking all the information you have just learned, it should be pretty obvious how the rocket's entity will behave when it enters the `Q3` world: Its movement is based entirely on its initial settings (such as position and velocity), and it doesn't think for 15 seconds. When 15 seconds are up, if the rocket is still intact, its `think` function is called—in this case, `G_ExplodeMissile`, which blows the rocket up and removes the entity from the level.

## NOTE

This is a relatively simple example of a `think/nextthink` relationship, and I guarantee that you will see some more complicated entities in the future, including ones where `think` functions tell the entity to point to new `think` functions!

## Entities: Building Blocks in Q3

Now that you have a handle on `think` and `nextthink`, let's go into more detail on just what an entity is. If you browse through any of the source files, you will see that entities come up a lot. For instance, in

the `fire_rocket` example, an entity is created and then defined as a rocket. You may be wondering: What is an entity? How many can you have? How do they behave?

Anything that is created in the game world is an *entity*; this includes players, bots, rockets, shotguns, ammo packs, animations of explosions, and much more. When a rocket needs to be created, a free entity is allocated and turned into a rocket. When it explodes or is removed, it is stripped of its properties and placed back in the pool.

There is currently a hard limit of 1,024 entities in the game; this is for all your objects, including players, bots, and weapons. This may not seem like a very large number, but if you think about it, it is far more than what's required. A typical *Q3* game consists of about 16 players, 40 weapon and ammo points, and a few powerups—let's say three. That's only 59 entities so far. Even if every one of the 16 players fired a rocket into the game, the total would only be pushed up to 75 entities. That means you are still left with more than 900 entities for plasma bursts, launched grenades, shotgun blasts, rail trails, and so on. Considering that an entity is used over and over again, and that it is deleted after a certain period of time, this number is more than sufficient.

### NOTE

If you want to investigate how entities are defined, open `g_local.h` in Header Files (part of the game project) and start reading from line 50. You should recognize a few things from when the rocket was created. I'll cover the variable type of an entity, `entity_t`, in greater detail in Chapter 3, "More Weaponry Work."

## Changing the Missile's Behavior

Now that you have seen how standard missiles work, you're ready to have a go at building a homing missile. Following is a description of how I would like this mod to work; I strongly recommend you write something similar for anything you develop. If it is a larger mod you're creating, then split it down into small chunks like this:

*The homing missile should be fired from the standard Q3 rocket launcher with the player being able to select either a standard or homing mode prior to launch. The missile should home in on its*

*nearest target, but have a wide turning circle so the player is able to dodge. The missile should also move more slowly than the standard rocket.*

Now that you have defined your mod, you can split the task into two separate elements:

- Getting input from the user to select homing or normal missile behavior
- Creating a homing missile that targets a player

## Getting the Input

To get input from the user, you'll create a console command that can be bound to a key. The console is the interface into which you can type direct commands, much like at a DOS prompt. You can toggle the console on and off by pressing the tilde (~) key on your keyboard.

Many activities that the player performs throughout *Q3*, such as jumping or firing a weapon, are assigned to console commands, and actual keys, joystick buttons, or mouse controls are set or bound to these console commands. Because you need to allow the player to specify whether he wants to fire normal missiles or homing missiles, you need to create a new console command for this. To do so, open `g_local.h` and go to line 227. You should see the code listing shown below.

```
1. typedef struct {
2.   clientConnected_t  connected;
3.   usercmd_t          cmd;           // we would lose angles if
   not persistant
4.   qboolean          localClient;    // true if "ip" info key is
   "localhost"
5.   qboolean          initialSpawn;   // the first spawn should be
   at a cool location
6.   qboolean          predictItemPickup; // based on cg_predictItems
   userinfo
7.   qboolean          pmoveFixed;
8.   char              netname[MAX_NETNAME];
9.   int               maxHealth;     // for handicapping
10.  int               enterTime;     // level.time the client
   entered the game
11.  playerTeamState_t teamState;     // status in teamplay games
```

```

12.  int          voteCount;          // to prevent people from
    constantly calling votes
13.  int          teamVoteCount;     // to prevent people from
    constantly calling votes
14.  qboolean     teamInfo;          // send team overlay
    updates?
15. } clientPersistant_t;

```

This creates a struct called `clientPersistant_t` that stores persistent client data across multiple respawns. In English: when a player is killed and then gets placed back into the level to continue playing, the variables in this struct will remain intact.

After line 14, add this line:

```
qboolean     homing_status;
```

Your last three lines should now look like this:

```

qboolean     teamInfo;          // send team overlay updates?
qboolean     homing_status;
} clientPersistant_t;

```

This creates a flag to indicate whether the homing missiles are on or off. Instead of using on and off, however, you use a variable called a boolean, which can be set to either true or false. The variable type `qboolean` is simply a new variable type that the programmers at ID created for use within *Q3*, and its values can either be `qtrue` or `qfalse`,

respectively. Next, you need to create a console command that sets this variable. Open `g_cmds.c` and scroll to line 1575. You should see a function starting with the following:

```

void ClientCommand( int clientNum ) {
    gentity_t *ent;
    char      cmd[MAX_TOKEN_CHARS];

```

At the end of this function, you should also see:

## NOTE

The `clientPersistant_t` struct is one of several very important properties of a player entity that enable it to maintain its state in various stages throughout the game. I will cover all these structs in more detail in Chapter 4, “Manipulating the Player.”

```

else if (Q_stricmp (cmd, "stats") == 0)
    Cmd_Stats_f( ent );
else
    trap_SendServerCommand( clientNum, va("print \"unknown cmd %s\n\"",
cmd ) );
}

```

Directly after the line that starts with `Cmd_Start_f`, insert the following:

```

else if (Q_stricmp (cmd, "homing") == 0)
    Cmd_SetHoming_f (ent);

```

This compares what has been entered on the command line with the word between the quotation marks. If a match is found, then the `Cmd_SetHoming_f` function is called. To add this function, head back up to line 1575, and insert the following code a line or so above the `void ClientCommand` function:

```

void Cmd_SetHoming_f (gentity_t *ent)
{
    if (ent->client->pers.homing_status == 1)
    {
        trap_SendServerCommand( ent-g_entities, va("print \"Homing Missiles
are off.\n\""));
        ent->client->pers.homing_status = 0;
    }
else
    {
        trap_SendServerCommand( ent-g_entities, va("print \"Homing Missiles
are on.\n\""));
        ent->client->pers.homing_status = 1;
    }
}

```

This checks whether the homing missiles are turned on; if so, they are switched to off, and vice versa.

The first bit of your mod has now been written. Now, let's create the homing missile.

## Seek and Destroy: Targeting a Player

The missile is going to search for a target within a certain radius. If a target is found, the missile will alter its direction to point at it. To find

a target within a certain radius, you need a new function. *Quake II* included this in its source, but it is missing from *Q3*; for this reason, you'll have to add it. Load up `g_utils.c` and add the following code to the end:

```
gentity_t *findradius (gentity_t *from, vec3_t org, float rad)
{
    vec3_t eorg;
    int j;
    if (!from)
        from = g_entities;
    else
        from++;
    for (; from < &g_entities[level.num_entities]; from++)
    {
        if (!from->inuse)
            continue;
        for (j=0; j<3; j++)
            eorg[j] = org[j] - (from->r.currentOrigin[j] + (from->r.mins[j] + from->r.maxs[j])*0.5);
        if (VectorLength(eorg) > rad)
            continue;
        return from;
    }
    return NULL;
}
```

Also, add this next bit of code after the `findradius` function:

```
qboolean visible( gentity_t *ent1, gentity_t *ent2 ) {
    trace_t      trace;
    trap_Trace (&trace, ent1->s.pos.trBase, NULL, NULL, ent2->s.pos.trBase, ent1->s.number, MASK_SHOT );
    if ( trace.contents & CONTENTS_SOLID ) {
        return qfalse;
    }
    return qtrue;
}
```

This traces a line from one entity to the other to see if it is visible. If the trace hits a wall, then `false` is returned; otherwise, it's true.

Next you need add the next two lines to the bottom of `g_local.h` to allow the `visible` and `findradius` functions to be called from another part of the program:

```
qboolean visible( gentity_t
*ent1, gentity_t *ent2 );
gentity_t *findradius (gen-
tity_t *from, vec3_t org,
float rad);
```

When this is finished, load up your `g_missile.c` file again and add the rather large segment of code in the next listing, just below the `#define MISSILE_PRESTEP_TIME 50` statement. This is your missile's think function. I've numbered the code so that it is easier to follow along with; you do not need to include the numbers in your actual code.

```
1. void G_HomingMissile( gentity_t *ent )
2. {
3.     gentity_t     *target = NULL;
4.     gentity_t     *rad = NULL;
5.     vec3_t     dir, dir2, raddir, start;
6.
7.     while ((rad = findradius(rad, ent->r.currentOrigin, 1000)) !=
        NULL)
8.     {
9.         if (!rad->client)
10.            continue;
11.        if (rad == ent->parent)
12.            continue;
13.        if (rad->health <= 0)
14.            continue;
15.        if (rad->client->sess.sessionTeam == TEAM_SPECTATOR)
16.            continue;
17.        if ( (g_gametype.integer == GT_TEAM || g_gametype.integer ==
            GT_CTF)
```

## TIP

`trap_Trace` is one of many system call functions built into Q3's code that are granted special access right into the 3D engine code—code that you do not get to see or modify.

These top-secret system calls are used frequently for very specific data that is retrieved in the most efficient ways possible. I'll be covering them in more detail as this book progresses, and will get into the specifics of `trap_Trace` in Chapter 6.



```
18.     && rad->client->sess.sessionTeam == rad->parent->client-
        >sess.sessionTeam)
19.         continue;
20.     if (!visible (ent, rad))
21.         continue;
22.
23.     VectorSubtract(rad->r.currentOrigin, ent->r.currentOrigin,
        raddir);
24.     raddir[2] += 16;
25.     if ((target == NULL) || (VectorLength(raddir) <
        VectorLength(dir)))
26.     {
27.         target = rad;
28.         VectorCopy(raddir, dir);
29.     }
30. }
31.
32. if (target != NULL)
33. {
34.     VectorCopy( ent->r.currentOrigin, start );
35.     VectorCopy( ent->r.currentAngles, dir2 );
36.     VectorNormalize(dir);
37.     VectorScale(dir, 0.2, dir);
38.     VectorAdd(dir, dir2, dir);
39.     VectorNormalize(dir);
40.     VectorCopy( start, ent->s.pos.trBase );
41.     VectorScale( dir, 400, ent->s.pos.trDelta );
42.     SnapVector( ent->s.pos.trDelta );
43.     VectorCopy (start, ent->r.currentOrigin);
44.     VectorCopy (dir, ent->r.currentAngles);
45. }
46. ent->nextthink = level.time + 100;
47. }
```

Line 7 dictates that when there are objects in a 1,000-unit radius, the code between the curly braces should be executed. The while loop is calling the `findradius` function you added earlier. Lines 9 and 10 state that if the target is not a player, the code sequence should continue. The `continue` statement breaks out of the while loop and starts again (the `findradius` function then moves on to the next entity).

The next several lines ask a series of questions. Line 11 asks whether it was the player who fired the weapon; line 13 asks whether the player is dead. Line 15 asks whether it was a spectator, while lines 17 and 18 try to determine whether the target is on the same team. Line 20 asks whether the target is visible. After all that, if the target is a valid one, the missile aligns itself to the target and moves forward. This is done with a series of vector manipulations. Finally, in line 46, the missile's next think time is set to 100.

What you must do now is modify your `fire_rocket` function a little. The next bit of code contains the complete version, which checks whether a homing or normal missile has been fired and sets the correct think function, damage, and velocity.

```
1. gentity_t *fire_rocket (gentity_t *self, vec3_t start, vec3_t dir) {
2.     gentity_t     *bolt;
3.
4.     VectorNormalize (dir);
5.
6.     bolt = G_Spawn();
7.     bolt->classname = "rocket";
8.
9.     if (self->client->pers.homing_status ==1)
10.    {
11.        bolt->nextthink = level.time + 60;
12.        bolt->think = G_HomingMissile;
13.        bolt->damage = 75;
14.        bolt->splashDamage = 100;
15.        bolt->splashRadius = 90;
16.
17.    } else {
18.        bolt->nextthink = level.time + 15000;
19.        bolt->think = G_ExplodeMissile;
20.        bolt->damage = 100;
21.        bolt->splashDamage = 100;
22.        bolt->splashRadius = 120;
23.    }
24.
25.    bolt->s.eType = ET_MISSILE;
26.    bolt->r.svFlags = SVF_USE_CURRENT_ORIGIN;
27.    bolt->s.weapon = WP_ROCKET_LAUNCHER;
```

```
28.     bolt->r.ownerNum = self->s.number;
29.     bolt->parent = self;
30.     bolt->methodOfDeath = MOD_ROCKET;
31.     bolt->splashMethodOfDeath = MOD_ROCKET_SPLASH;
32.     bolt->clipmask = MASK_SHOT;
33.     bolt->target_ent = NULL;
34.     bolt->s.pos.trType = TR_LINEAR;
35.     bolt->s.pos.trTime = level.time - MISSILE_PRESTEP_TIME; //
    move a bit on first frame
36.     VectorCopy( start, bolt->s.pos.trBase );
37.
38.     if (self->client->pers.homing_status ==1)
39.         VectorScale( dir, 400, bolt->s.pos.trDelta );
40.     else
41.         VectorScale( dir, 900, bolt->s.pos.trDelta );
42.
43.     SnapVector( bolt->s.pos.trDelta ); // save net bandwidth
44.     VectorCopy (start, bolt->r.currentOrigin);
45.
46.     return bolt;
47. }
```

Lines 9 through 23 are new, having been moved from their original spot between lines 29 and 30. Lines 38 through 41 are also new, slowing the missile down if it is a homing missile. There is one last thing you need to do to complete this mod: Set the homing missile to be off by default. Load `g_client.c` and go to line 966; the code you'll see there is outlined below.

```
void ClientBegin( int clientNum ) {
    gentity_t     *ent;
    gclient_t     *client;
    gentity_t     *tent;
    int           flags;

    ent = g_entities + clientNum;

    client = level.clients + clientNum;
```

This function is called after a client connects and enters the level and sets a few parameters.

Add the following line after the preceding code:

```
client->pers.homing_status = 0;
```

That's it! You're finished. Compile the mod into a DLL and run it. To test your homing missile, you will need to bind a key to the homing toggle you wrote earlier. Do this by bringing up your console with the ~ key and typing /bind h homing. Fire away.

## Smoothing the Missile

I trust you have had a little play with the homing missile, and have probably noticed that its movement is a little jerky. Here is a little fix that smoothes its movement to give a more realistic feel. Place the code below after the homing missile's think function.

```
void Missile_Smooth_H( gentity_t *ent, vec3_t origin, trace_t *tr ) {

    int touch[MAX_GENTITIES];
    vec3_t mins, maxs;
    int num;

    num = trap_EntitiesInBox( mins, maxs, touch, MAX_GENTITIES );
    VectorAdd( origin, ent->r.mins, mins );
    VectorAdd( origin, ent->r.maxs, maxs );
    VectorCopy( origin, ent->s.pos.trBase );
    ent->s.pos.trTime = level.time;
}
```

Then, in the `G_RunMissile` function, add the following after `VectorCopy( tr.endpos, ent->r.currentOrigin )`:

```
Missile_Smooth_H(ent,origin,&tr);
```

Your code now looks something like this:

```
else {
    VectorCopy( tr.endpos, ent->r.currentOrigin );
}
Missile_Smooth_H(ent,origin,&tr);
trap_LinkEntity( ent );
```

Run your mod again; you'll see the missile smoothly chasing its target. Figure 2.4 shows one of the new missiles locking in on a *Q3* bot.



**Figure 2.4** A homing missile fired to the left adjusts its path accordingly.

## A Final Note

There are probably a couple of things you could do to enhance this mod. For example, normal missiles are destroyed after 15 seconds, whereas the homing missiles carry on until they hit something. This could present a problem if missiles are fired into the air and don't locate a target, because they would go on forever. It would be relatively simple to add a counter to the `think` function and destroy the missile after a certain period of time.

## Summary

I hope this chapter has given you some insight into how *Q3* works and encouraged you to jump in and make changes. From here on in, I'm going to be very specific in how you implement changes, breaking down variable type definitions, and the mechanics of each function that is used. This way, you'll start to understand what is really happening in the code, giving you the power to learn and grow with mod development on your own.



# **CHAPTER 3**

# **MORE WEAPONRY WORK**

It may not seem like much, but you've successfully built a mod! By changing the behavior of the rocket, you've systematically altered the way *Q3* will handle the rules of its world. Many mods start out this way—a simple tweak here and a change there, eventually evolving into thousands of lines of code that will ultimately change *Q3* into an entirely new game.

Because you have the weapon code from `g_weapon.c` fresh in your mind, let's spend this chapter further exploring this integral part of the game. You'll learn about the various weapon types, and how you can modify them to better suit your needs.

## Understanding Weapon Types

Because *Q3* falls into the “first-person shooter” category, it should come as no surprise that one of its most important features is the way it handles weapons. A quality first-person shooter (henceforth to be known as an *FPS*) game offers players a considerable arsenal of weapons designed to cause destruction and mayhem. Many FPS games are noted for their unique weapons, or weapons that are modeled closely to match their real-life counterparts. Ultimately, it will be up to you to decide how weapons will behave in your mod, however realistic or imaginary they may be.

Although *Q3* has nine individual weapons, they can be broken down into one of the four following categories:

- **Hitscan weapons.** A *hitscan weapon* is so named because it performs a scan of a target area and calculates the total amount of damage based on the accuracy of the hit. These weapons generally do not have visible projectiles (such as bullets or missiles) unless they are used for client effects such as the ejection of a spent shell. They are also often fired in bursts or heavy

repetition. Examples of hitscan weapons include the shotgun and the chaingun (found in the Team Arena Expansion pack).

- **Missile weapons.** Weapons in this category generally cast out a visible projectile of some sort, such as a rocket or a grenade. Typically, they do not fire in rapid succession. The missile itself causes two types of damage: direct damage (from making full contact with another surface or enemy) and splash damage (damage inflicted by the explosion of the missile). Generally, splash damage varies in intensity based on where the target lies on the explosion's radius—that is, the distance the target is from the explosion's center point.
- **Beam weapons.** A *beam weapon* is one that produces a constant barrage of ammunition toward a given target. *Q3's* lightning gun is a perfect example of a beam weapon, as is the flamethrower in *Return to Castle Wolfenstein* (see Figure 3.1). Beam weapons generally release ammunition at an extremely fast pace and deal out the most damage, because they often involve a fair amount of accuracy on the part of the shooter. Beam weapons, however, typically have less range than missile weapons.



**Figure 3.1** *The flamethrower in Return to Castle Wolfenstein.*



- **Melee weapons.** *Melee weapons* are often overlooked, but are nonetheless a vital weapon in any arsenal. Indeed, the melee weapon is a player's last-ditch chance at survival if all other forms of ammunition are exhausted. A melee weapon is simply one that is held by the player

and swung or thrust toward an enemy. The most common types of melee weapons are axes, bats, crowbars, and the like. *Q3* has one melee weapon, the gauntlet, which acts like an electrified fist used to punch opponents if they are within range.

If you understand each of these weapon categories before creating your mod, you will have a much better idea of what types of specific weapons you will want to implement in your mod. Weapon familiarity will be a fundamental building block for you in your quest to become a mod programmer.

## Modifying the Shotgun

One of the easiest weapons to work with in *Q3* is the shotgun, so let's start there. As you may know by playing around with *Q3*, the shotgun is a fairly straightforward weapon. Each time it is fired, a burst of shots is blasted toward a given target, each creating a slightly different pattern than the previous. Before you start changing the shotgun's behavior, however, let's take some time to learn how it works.

## Understanding the Top-Down Approach

Anytime you work with any type of object in code, you'll typically "drill down" from the most general aspects to the most specific parts of a given idea. This approach is known as the *top-down approach*, and is commonly used even beyond the scope of game programming. The

### NOTE

*Return to Castle Wolfenstein* is a sequel to id Software's first great 3D game, *Wolfenstein 3D*, which predates even *DooM*. Even though another company obtained the license to create *Return to Castle Wolfenstein*, guess what 3D engine drives the game? Yep, you guessed it: The *Quake III* engine.

top-down approach is so widely used, in fact, that it forms the basis of other programming languages, such as C++ and Java. Understanding the top-down approach is essential to working with *Q3*'s code base, as you shall see with the code to support *Q3*'s shotgun.

To give you a concrete example of the top-down approach, here is how it applies to the shotgun code: When a player in *Q3* fires the shotgun, the game code calls a function that fires the shotgun, which in turn calls a function that determines the pattern of bullets that burst forth when the shotgun is fired. This function then calls a third function, which calculates each unique bullet's path. You can see from this simple example that you are moving from the most general (fire the gun) to the most specific (where each bullet goes).

## Knowing the Shotgun Inside and Out

To gain an understanding of how the shotgun works, start by opening `g_weapon.c` and scrolling down to line 233 or so. You should see a comment header alerting you that you're entering a chunk of code that will handle the shotgun's behavior in *Q3* (it's the giant C-style comment with the word `SHOTGUN` surrounded by `= signs`).

As you can see, the first function is `ShotgunPellet`, which is called by the second function `ShotgunPattern`, which in turn, is called by the function `weapon_supershotgun_fire`. Let's take a look at what goes on behind the scenes in these three functions. The functions are ordered in the file in the reverse of the top-down approach (`ShotgunPellet` is the most specific), simply because it is required by C to define a function before it is called (unless explicitly declared in a header file).

The function declaration of `weapon_supershotgun_fire` looks like so:

```
void weapon_supershotgun_fire
(gentity_t *ent)
```

The function's return type is `void`, which means it will return nothing upon completion. It also requires one input parameter, a variable of type `gentity_t`, which is a user-defined data type that the

### NOTE

I will revisit the `gentity_t` data type many times throughout the course of this book, so I'll get to the actual meat of the code here instead of discussing `gentity_t` in more detail.

programmers at id created to refer to all the various types of entities that appear in the game.

In a nutshell, the `weapon_supershotgun_fire` function creates a temporary entity, which holds the flash of the shotgun's muzzle. This entity's position is then slightly moved or *scaled* forward from the muzzle to appear as though the weapon in question is producing it. A random *seed* is then created and assigned to one of the event parameter properties an entity may hold, which helps create the effect of the random spread of bullets that one might expect a shotgun to produce.

At the end of the `weapon_supershotgun_fire` function, you see `ShotgunPattern` is called. Its function call looks like this:

```
void ShotgunPattern( vec3_t origin, vec3_t origin2, int seed,
    gentity_t *ent )
```

Like `weapon_supershotgun_fire`, `ShotgunPattern`'s return type is `void`, which means it will return nothing upon completion. As for what you need to pass into this function, the inputs consist of a `vec3_t`, a second `vec3_t`, the random *seed* generated earlier, and the `gentity_t` passed into `weapon_supershotgun_fire`. (I'll get to what exactly `vec3_t` and `gentity_t` variables are in a moment.)

The brunt of the `ShotgunPattern` function, however, is dedicated to the firing of each individual bullet, testing for a hit on the target from that bullet, and if a successful hit is found, accruing the attacker's accuracy variable.

Within a loop of 11 bullets near line 319 in `g_weapon.c`, each bullet's position is randomly generated, based on the firing origin. Then, the third function, `ShotgunPellet`, is called. The function call looks like this:

```
qboolean ShotgunPellet( vec3_t start, vec3_t end, gentity_t *ent )
```

## NOTE

In real life, shotgun ammunition usually comes in the form of a *gauge*, such as 12 or 20 gauge, which itself is a plastic shell containing small bullets or *buckshot*. When fired, the ejection of the bullet causes the casing to explode, sending the buckshot out in a randomly spread pattern toward its target. In essence, no two shots fired from a shotgun should produce the same effect. By creating this random *seed*, Q3 is attempting to reproduce a shotgun's randomness.

**NOTE**

Players in *Q3* have multiple variables associated with them that indicate how the player has been performing during play. An example of this is the `accuracy` variable, which is a direct indication of how many shots that have been fired actually hit their target. There are also variables to indicate how many shots a player has fired, how much damage has been taken by his armor, how long he has been idle in the game, and more. All these variables are a part of the `gclient_s` struct, which is declared around line 240 in `g_local.h`. You will get to know the `gclient_s` struct in greater detail in Chapter 4, “Manipulating the Player.”

This function takes three parameters: a `vec3_t` to start, a `vec3_t` to end, and a *Q3* entity to which the effects of the bullet will be applied (I think you can take a wild guess that the entity in question will be the target). The function returns a variable that is of type `qboolean`. A *Boolean* variable is simply a variable that can evaluate to true or false; on and off, yes and no, and 1 and 0 are all examples of Booleans. The programmers at ID Software simply created their own Booleans, but rest assured, you are free to use them as freely as they did to return values that can either be true or false. And if you guessed that a `qboolean` returns `qtrue` or `qfalse`, you can award yourself an extra 50 points right now.

## The Physics of Vectors

You had a bit of exposure to the `vec3_t`, or *vector* variable type, when you played with the homing-missile code in Chapter 2. My advice is: Get used to it! Vectors are used frequently in the *Q3* code base; they're a fundamental building block to representing the data of a three-dimensional world. In that respect, a quick definition of the vector is in order. In a nutshell, a *vector* is a measurement by which to gauge *in which direction* or *how far* a particular item is from a given point. For its application in *Q3*, the vector's data type consists of three important numbers: a value for the x axis, a value for the y axis, and a value for the z axis.

However, three arbitrary numbers just don't "cut the mustard," as there is nothing to reference those values from. To obtain real data from a vector in *Q3*, you reference the *left-hand coordinate system*, on which *Q3*'s world is based. What this means is, given a specific point in the world, values above the point, to the right of the point, and further away (imaging a jet flying overhead, towards the horizon) from the point are *positive* values of x, y, and z. As expected, values below (literally, as in underneath) the point, to the left of the point, and away from the horizon (meaning towards you) are *negative* values of x, y, and z. Knowing the values held in a vector, and comparing them to a position in the left-hand coordinate system allows usable data to be extracted from a vector, including *position* and *distance*.

Internally, the `vec3_t` variable type that was created for *Q3* is simply an array of floats. A *float*, if you recall, is a numerical data type that can have a decimal value, such as 1.5, or 6.87354. An *array* is a complex data type in C, which can contain multiple values that are referenced by an index, or common key. Because the `vec3_t` data type must hold a value for x, y, and z in 3D space, it is a three-dimensional array, which looks like `var[0]`, `var[1]`, and `var[2]`, where `var` is the float. (Arrays always start at 0 in the C Language.) Feel free to look up the declaration of `vec3_t` in `q_shared.h`, near line 452. Since all vectors are relative, the *Q3* code describes their relativity with the variable names *forward* (describing the value in `vec3_t[0]`), *right* (the value in `vec3_t[1]`), and *up* (the value in `vec3_t[2]`.)

## Intricacies of Damage

Diving into the actual code within the `ShotgunPellet` function, you can discern a number of interesting tidbits. First, this is the function that actually calculates what entity the bullet hits. It does this with a call to `trap_Trace`, a system call function that passes some data directly into *Q3*'s executable for a resolution.

You may recall that `trap_Trace` was used in the homing-missile code; it is a commonly used function that will pop up often throughout this book. For now, all you need to remember is that the `trap_Trace` function will take a start and end point, and draw an invisible line or *trace* to see if anything interfered with the line. What would interfere with the line, you ask? How about a player trying to block your shotgun fire with his face? All kidding aside, the results of `trap_Trace` will indicate

to `ShotgunPellet` whether the bullet actually came into contact with something that is capable of registering a hit. (A box of ammunition, for example, is inanimate and bullets pass right through it. Of course, you might want to change that . . . )

If the entity has a valid surface (indicated by a property called `surfaceflags`), the function then goes on to calculate damage that the bullet would cause. You will see that on or around line 267, the damage is calculated as so:

```
damage = DEFAULT_SHOTGUN_DAMAGE * s_quadFactor;
```

The `DEFAULT_SHOTGUN_DAMAGE` variable is defined at the top of the shotgun's code in `g_weapon.c` as 10. Changing this variable is the quickest way to increase or decrease the amount of damage each shotgun bullet causes. Notice also that the damage is multiplied by the variable `s_quadFactor`, which is used to hold the value of the player's Quad Damage powerup (if he happens to be carrying that powerup at the time of firing).

Finally, if all signs indicate that the bullet hit a surface that can be damaged, the damage is dealt out with a call to `G_Damage` near line 290. `G_Damage` is called like this:

```
void G_Damage( gentity_t *targ, gentity_t *inflictor, gentity_t
*attacker, vec3_t dir, vec3_t point, int damage, int dflags, int mod )
```

`G_Damage` is a function that returns no data, but requires a slew of input parameters:

- **targ.** The entity in the game that will receive the damage.
- **inflictor.** The entity that caused the damage, in a literal sense. This could be a bullet, a grenade, a rocket, and so on.
- **attacker.** The entity that caused the inflictor to do damage. An example is the player who fired the shotgun.
- **dir.** This is the direction of “knock back” for the target. Some items cause so much damage that the target may be thrown back from the blow.
- **point.** This is the point of actual contact where the damage is done. This is a good reference point in case you want to look for damage in specific areas on the target, such as a headshot.
- **damage.** Simply, the total amount of damage being dealt to the target.

- **dflags.** Some types of damage can be specialized or *flagged* to behave differently within *Q3*. This is where the damage flags are passed in. The available *dflags* are:
  - **DAMAGE\_RADIUS:** the damage was applied indirectly, such as from an explosion.
  - **DAMAGE\_NO\_ARMOR:** the damage ignores player's armor
  - **DAMAGE\_NO\_KNOCKBACK:** the damage doesn't affect a player's velocity.
  - **DAMAGE\_NO\_PROTECTION:** there is no protection against this damage (not even god-mode, which normally makes a player invincible!)
  - **DAMAGE\_NO\_TEAM\_PROTECTION:** used specifically to identify Team Arena's "Kamikaze" massive supernova that kills everything in a wide radius.
- **mod.** These are flags that tell the function what will be considered the means of death in case it is to be determined at a later point (for example, if you want to track a certain weapon's usage and effectiveness). Chapter 6, "Client Programming," will cover the means-of-death flags in greater detail.

A final function, `LogAccuracyHit`, is called to perform an accuracy log calculation—think of it as a final sanity check to make sure the attacker and the target are both valid. The function then returns with either a `qtrue` or `qfalse`, depending on the success of the shot.

Now that you have an understanding of the how the shotgun works, you can begin to make adjustments to how it performs.

## Adjusting the Shotgun's Accuracy

In the real world, running and accurately firing a shotgun at the same time would be quite difficult. That lack of accuracy, however, is offset by the fact that a shotgun's spread has a larger area of coverage than a generic pistol or handgun. Still, it makes sense that a person standing motionless would have better luck aiming and firing than a person running backward at full speed. Better yet, a person crouching would have the additional balance to aim with even greater skill. With that in mind, let's set how to improve the shotgun's accuracy if the player is crouching, and lessen the accuracy if he is on the move.

The first piece of information you need to know in order to implement this change is how you can tell if the player is crouching. By conducting a simple search across the *Q3* code tree, you will discover various references to a flag called `PMF_DUCKED`. `PMF_DUCKED` belongs to a larger subset of flags, called *pmove flags*, which are used to describe how the player is moving in the game. (As you learned in the previous section, a *flag* is simply a type of variable that you can attach to a given entity in order to make the entity behave or perform differently within the game.) Table 3.1 lists the *pmove* flags in *Q3* and what they stand for:

The `PMF_DUCKED` flag is the variable you check to see if the player is in a crouching or *ducked* state. In addition to being able to determine whether the player is crouching or standing, you want to be able to scale a factor of accuracy up or down based on this state. To do so, you create an integer variable called `accuracyFactor`, which will be used in your function modification.

Armed with that knowledge, let's go do some damage in the code! Start by moving to the shotgun's middle function, which you'll recall is `ShotgunPattern`. After the declaration of `hitClient` (around line 308), add the following code:

```
int      i;
float    r, u;
vec3_t   end;
vec3_t   forward, right, up;
int      oldScore;
qboolean hitClient = qfalse;
int      accuracyFactor = 4; // Our default accuracy multiplier is 4
```

**TIP**  
 You can search across the *Q3* code tree by opening the Edit menu from within Visual Studio and choosing the Find in Files command. This opens a dialog box in which you can search for a specific string of text by typing it in the Find What text box. You can also specify the folder in which to start your search by changing the folder path in the In Folder text box. To search across the entire *Q3* code base, change the In Folder path to `C:\quake3\code\`, where `C:` equals the drive in which you installed the source.



**Table 3.1** pmove Flags

Variable	Value
PMF_DUCKED	This flag indicates that the player is crouching.
PMF_JUMP_HELD	This flag indicates that the player is holding the jump button down.
PMF_BACKWARDS_JUMP	This flag indicates that the player is jumping backwards.
PMF_BACKWARDS_RUN	This flag indicates that the player is running backwards.
PMF_TIME_LAND	This flag indicates that the player is landing from a jump, typically used to tell Q3 when the player is allowed to jump again.
PMF_TIME_KNOCKBACK	This flag indicates that the player is recoiling from damage.
PMF_TIME_WATERJUMP	This flag indicates that the player is jumping in water.
PMF_RESPAWNED	This flag indicates that the player has respawned in the level.
PMF_USE_ITEM_HELD	This flag indicates that the player is using a held-item (such as the personal teleporter).
PMF_GRAPPLE_PULL	This flag indicates that the player being pulled by the grappling hook (a weapon removed from the released version of Q3).
PMF_FOLLOW	This flag indicates that the player is actually spectating another player in the game.
PMF_SCOREBOARD	This flag indicates that the player is spectating as a scoreboard (not spectating any particular player).
PMF_INVULEXPAND	This flag indicates that the player is surrounded with Team Arena's Invulnerability Sphere.
PMF_ALL_TIMES	This flag is a combination of PMF_TIME_WATERJUMP, PMF_TIME_LAND, and PMF_TIME_KNOCKBACK.

Note the new variable, `accuracyFactor`, at the end of the declaration list; it will be responsible for dictating the accuracy of a player's shot (higher values meaning less accurate). Next, scroll down to the line of code that copies the player's current score into a local variable. This is an excellent place for you to add player crouch detection. Make the following code adjustments:

```
oldScore = ent->client->ps.persistent[PERS_SCORE];
// is the user crouching? bump up the accuracy!
if (ent->client->ps.pm_flags & PMF_DUCKED)
    accuracyFactor = 1;
```

Here, you indicate in the code that if the player's `pmmove` flags contain the flag `PMF_DUCKED`, the accuracy factor should be changed from 4 to 1.

Once you have your updated `accuracyFactor` variable, you can apply it to the actual code that generates the shotgun spread when fired. Scroll down to the following code chunk in `ShotgunPattern`, near line 320, and make the noted changes:

```
for ( i = 0 ; i < DEFAULT_SHOTGUN_COUNT ; i++ ) {
    r = Q_crandom( &seed ) * DEFAULT_SHOTGUN_SPREAD * accuracyFactor * 16;
    u = Q_crandom( &seed ) * DEFAULT_SHOTGUN_SPREAD * accuracyFactor * 16;
    VectorMA( origin, 8192 * 16, forward, end);
    VectorMA( end, r, right, end);
    VectorMA( end, u, up, end);
```

Looking at the lines of code that calculate the starting *right* and *up* positions (as noted by the `r` and `u` variables, respectively), you are now multiplying the variable `DEFAULT_SHOTGUN_SPREAD` by your noted `accuracyFactor` variable. Because the `accuracyFactor` variable's default value is 4, and is only ever modified if the player crouches, the shotgun will behave exactly as it does in regular *Q3* if the player crouches first.

## NOTE

The reason you check whether the player's flags *contain* `PMF_DUCKED` as opposed to *actually equaling* `PMF_DUCKED` is simple: The very nature of flag usage in *Q3* is such that they can be mixed and matched, added or removed, without any particular dependency upon one another. Thus, at any given time in the game, the player's `pmmove` flags may contain `PMF_DUCKED` along with any number of other flags.

### Working a Bit at a Time

All the various types of bit flags that are used in *Q3* require the use of *bitwise operators*. Unlike standard logical operators *and* (&&), and *or* (||), which evaluate an expression to either *true* or *false*, bitwise operators work at the binary level, comparing the bits of a given variable to those of another variable. Explanations of bitwise operations can become very complicated very quickly, so here's what you need to know for now: In *Q3*, when you check for a flag being turned on, you use the *bitwise and operator* (&), like so:

```
if (ent->client->ps.pm_flags & PMF_DUCKED)
```

You may also group flags together using the *bitwise or operator* (|). Although it sounds confusing, trust me: associating flags together with *bitwise or* is the same as saying “this flag *and* that flag.” For example, to check whether the player is holding the Jump button down *and* is jumping backwards, you'd use the following:

```
if (ent->client->ps.pm_flags & (PMF_JUMP_HELD |  
PMF_BACKWARDS_JUMP ))
```

With bitwise logical operators, you can also check whether certain flags are missing by using the *inverse bitwise operator* (~). For

If you're confused, let me clarify: The shotgun's `DEFAULT_SHOTGUN_SPREAD` variable, multiplied by 1, will always be its original value (which in *Q3* is 700). This is the effect that takes place when your player crouches and fires the newly modified shotgun. Standing and firing the gun will cause a greater spread, since `DEFAULT_SHOTGUN_SPREAD` is multiplied by 4 instead of 1.

## The Shotgun's Dirty Secret

Your mod of the shotgun's firing accuracy is not quite complete. If you were to compile and run it now, everything would seem in order. A quick observation of the intended change, however, would reveal

example, to check whether a player is *not* crouching, you'd use the following:

```
if (ent->client->ps.pm_flags & ~PMF_DUCKED)
```

When it comes time to start adding or removing bit flags from your variable, you'll use the *bitwise or assignment operator* (`|=`). For example, to add the knock-back pmove flag, you'd use the following:

```
player->client->ps.pm_flags |= PMF_TIME_KNOCKBACK;
```

To remove a bit flag, use the *bitwise and assignment operator* (`&=`), and invert the flag with the `~` symbol. For example, to remove the `PMF_TIME_KNOCKBACK` flag you just added, you'd do the following:

```
player->client->ps.pm_flags &= ~PMF_TIME_KNOCKBACK;
```

Finally, to toggle a bit flag, regardless of its current status (either on or off), use the *bitwise exclusive or assignment operator* (`^=`). For example, to toggle the player crouched flag on (if it is off) or off (if it is on), use the following:

```
ent->client->ps.pm_flags ^= PMF_DUCKED;
```

There are many more types of bitwise operations and assignments, but for the most part, everything you will need to do in Q3 is listed above.

nothing new. Standing or crouching, if you were to fire the shotgun into a wall, you would see the same basic pattern as before.

So what's the deal? When you modified the rocket in Chapter 2, all you had to do was make a few changes in a few files, compile, and run. The difference here is that the effects of the shotgun, unlike those of the rocket, do not reside in the game module of the code. Take a deep breath, because this is where things get really exciting.

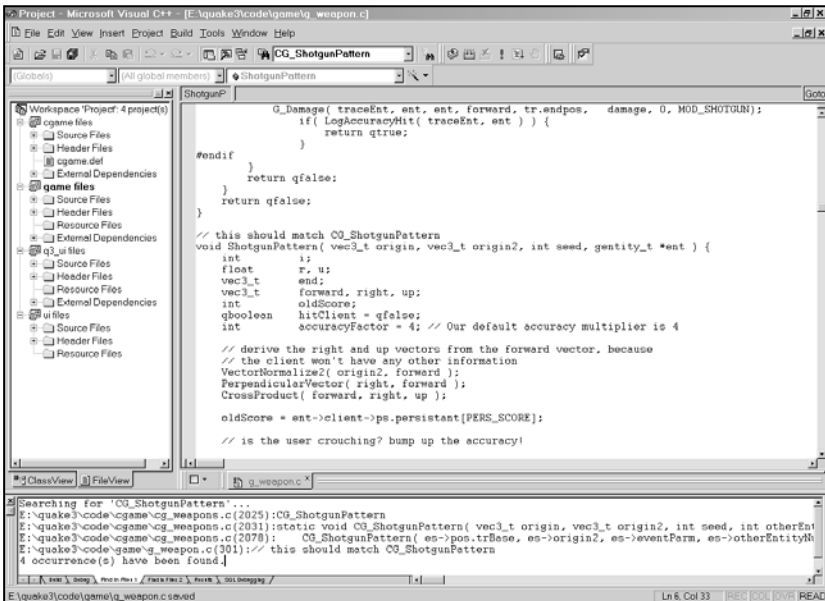
Look at the top of `ShotgunPattern` function. You'll see a C-style comment that the programmers at id Software left for you as a clue:

```
// this should match CG_ShotgunPattern
```

Hmm, `CG_ShotgunPattern` isn't a function you've seen yet. If you do a quick search across all of the *Q3* source code, you can see that `CG_ShotgunPattern` comes up in a file called `cg_weapons.c`; this is *definitely* not a file you've worked with. In fact, by looking at the Find dialog in VC++'s interface, near the bottom of the editor as shown in Figure 3.2, you can see that `cg_weapons.c` is in a different folder than the one in which you have been working, the `\cgame\` folder. If you double-click on the first occurrence of the string "`cg_weapons.c`" in the Find dialog, the file will pop into view. Sure enough, around line 2025, you can see a comment that introduces the `CG_ShotgunPattern` function; right below the comment, the function is defined:

```
static void CG_ShotgunPattern( vec3_t origin, vec3_t origin2, int seed,
int otherEntNum )
```

At first glance, `CG_ShotgunPattern` looks very similar to the first `ShotgunPattern` function you worked with, although it does return a static void instead of just a plain-old void type. Returning a static void is really the same as returning a normal void—but in reality, neither function returns anything! The difference here is that by declaring



**Figure 3.2** Searching for `CG_ShotgunPattern` in Visual Studio

the function static, you cause the function to be seen only by other functions in its file (in this case, cg\_weapons.c). As for the input parameters, there are still the two vector origin points and the random seed. Instead of a gentity\_t type for the fourth parameter, however, you see a simple integer type called otherEntNum. You will see how this comes into play in just a moment.

By looking at the first few lines of the CG\_ShotgunPattern function, you can see that it behaves in much the same manner as the ShotgunPattern function. VectorNormalize2 is called, then PerpendicularVector, followed by CrossProduct. There is even a loop to generate the random spread of bullet fire in much the same way as was performed in ShotgunPattern.

Ack! What's going on? Why do you need two different copies of essentially the same function? What is cgame code, and how is it different from game code? Why didn't you have to mess with the cgame code when you modified the rocket's behavior?

Whoa, whoa, whoa, let's take it one step at a time. Here are the facts: The shotgun's behavior within the world of *Q3* necessitates the need to store its control half in the server-side portion of the game code and half in the client-side portion. This is mostly due to a judgment call made by ID to decide what could be sacrificed to the client in order to save bandwidth when playing online. If you're confused already, let me put your mind at ease: I will cover the topic of cgame code, bandwidth, server-side and client-side code, and much more later in this book. For now, trust me. You will simply work with the cgame code, based on the assumption that the shotgun's effects are better suited to the client-side code.

So, now that you trust me (and thanks for waiting this long!), how can you go about making this change? As I stated earlier, the gentity\_t parameter is not going to be available to you within the scope of this

**NOTE**  
 The function that actually fires the bullet in this file ends up, predictably, being called CG\_ShotgunPellet. I'll leave it up to you to look at the contents of CG\_ShotgunPellet, if you dare. I'll warn you now: It's nothing like the ShotgunPellet function (but it is sitting right above CG\_ShotgunPattern, around line 1973).

function, so how do you determine whether the player is crouching? Fortunately, there is a way to tell what the player is doing with some client-side trickery.

## Synchronicity in the Client Code

On the client side of things, if you have an integer to work with (`otherEntNum` being your integer in this case), you can compare it to a property found in a variable that is held on the client called `cg`. `cg` is a variable that pops up from time to time as you peruse the `cgame` code. It is a very important variable, because along with a few others, it ultimately leads you to one very important person: the player.

Unlike server code, which must deal with many different entities and players working with (and against) each other, the client code needs to worry about only one entity: the person currently viewing the screen. Fortunately, `cg` has a property you can look up that will map to some of the player's server-side properties, such as whether the player is ducking. This property is `cg.snap`. The `snap` property then points to a struct you have access to, called `ps` (short for "player state").

Hmmm, that sounds familiar . . . yes! Sure enough, you used `ps` in your `ShotgunPattern` function! Once you have access to the `ps` struct, you have access to a further set of properties, including `clientNum`, which is also an integer. You can compare `otherEntNum`'s value to `clientNum` to see if you're dealing with the proper client entity.

Armed with that knowledge, let's make the appropriate adjustments. Add your `accuracyFactor` declaration near the top of `CG_ShotgunPattern` like so:

```
vec3_t      end;
vec3_t      forward, right, up;
int         accuracyFactor = 4; // same as game code!
```

Then, add this code right after the call to `CrossProduct`:

```
CrossProduct( forward, right, up );

// is the user crouching? bump up the accuracy!
if ( (otherEntNum == cg.snap->ps.clientNum) && (cg.snap->ps.pm_flags & PMF_DUCKED) )
    accuracyFactor = 1;
```

Finally, add the `accuracyFactor` multiplier to this spread function in exactly the same manner as you did in `ShotgunPattern`:

```
for ( i = 0 ; i < DEFAULT_SHOTGUN_COUNT ; i++ ) {
    r = Q_crandom( &seed ) * DEFAULT_SHOTGUN_SPREAD *
accuracyFactor * 16;
    u = Q_crandom( &seed ) * DEFAULT_SHOTGUN_SPREAD *
accuracyFactor * 16;
    VectorMA( origin, 8192 * 16, forward, end);
```

Excellent. With this new client-side code update, your mod has all the ingredients to make the shotgun behave differently. On the server side of things, your modification to `ShotgunPattern` will allow the server to properly calculate where shotgun blasts are to be employed against other entities in the game. And, on the client side of things, your new code will make sure that your shotgun's firing pattern truly does look different when fired from a crouching position than when fired while standing. These mutual changes in both `game` and `cgame` should maintain synchronicity between the two distinct modules. Let's compile the code and give it a try.

After compiling your changes, you should have two DLLs: `qagamex86.dll` and `cgamex86.dll`. Make sure both of these files are copied over to your `MyMod` directory. You also need to run *Q3* with the `sv_pure` value set to 0. This forces *Q3* to run as a standard server instead of as a pure server (see the ensuing sidebar for more information about standard servers and pure servers). To refresh your memory, here is a command line to type:

```
quake3.exe +set fs_game MyMod +set sv_pure 0 +map q3dm1
```

Sure enough, with a few tests you can see that when fired, the shotgun has a much wider spread while the player is standing, but while the player is crouching, the blast is more focused (see Figures 3.3 and 3.4).

There is some serious spread happening in the shot in Figure 3.3—maybe a bit too much. At least by looking at these two images, you can clearly see that you have made a significant change in the behavior of the shotgun. Now that you have a visual indication of just how much you affected the shotgun, you can begin to roughen out the edges and make the shotgun perform in an even more realistic manner.



### A Pure Server Is a Virtuous Server

One of the features id Software has had built into *Quake* since *Q1* is its ability to allow clients to download new files, right from within the game. This is so that if a player joins a server online that is running a new map or has new weapon models, those new data files can be downloaded immediately. That allows the player to begin play right away, without having to go look for them on an Internet Web site.

Unfortunately, some players who used demo *Quake* clients that they downloaded free from the Internet were dynamically downloading the official copyrighted *Quake* maps by connecting to servers running legitimate copies! Because that was not a very good business solution for id Software, the company built a server setting into *Q3* called *pure server*. If a *Q3* server runs with the pure server setting turned on, the client's files *must exactly match* the files on the server. So if you're using any new files, such as DLLs, or new maps or weapon models, and you don't launch *Q3* with the pure server setting disabled (`sv_pure 0`), *Q3* will ignore any extraneous files when it loads.

## Adding Polish: Shooting While Moving

Since you already have a good idea how to extract player information to modify the shotgun's behavior, let's go back and see if you can extrapolate data relating to the player's movement. Jump back into `g_weapon.c`'s code, and scroll down to where you originally did your code change for `ShotgunPattern` (near line 302). Start by adjusting the `accuracyFactor` a bit; set the default at 2 now.

```
int      oldScore;
qboolean hitClient = qfalse;
int      accuracyFactor = 2;
```



Figure 3.3 Your shotgun's fire while standing ...



Figure 3.4 ... and while crouching.

The ducking code can be left alone, because it performs correctly as is. Right below it (near line 321), add the following code:

```
// is the user moving? lower the accuracy!  
    else if (ent->client->ps.velocity[0] || ent->client->ps.velocity[1])  
        accuracyFactor = 3;
```

Once again, you fall back on the player state of the client, or `ent->client->ps`. Like the `pm_flags` property found within `ps`, another property available to you is `velocity`, which is a `vec3_t` data type that holds values pertaining to the player's movement. Because the variable `velocity` is of type `vec3_t`, and you know from experience that `vec3_t` is actually an array, you can access the three elements of `velocity` by their index, either `velocity[0]` for forward/backward movement, `velocity[1]` for left/right movement, and `velocity[2]` for up/down movement (like jumping or falling).

Because you are already modifying the shotgun's accuracy if the player is moving, let's take it one step further. For fun, let's drop the shotgun's accuracy even more if the player is found to be jumping or falling—that is, if the player is moving along the *z* axis:

```
// is the user jumping? lower it more!  
    else if (ent->client->ps.velocity[2])  
        accuracyFactor = 4;
```

There it is! A fully modified shotgun . . . well, enough so that its behavior more closely mimics a person's ability during a heated firefight.

## NOTE

I suppose if you wanted to get technical, you could argue that a shotgun's spread is ultimately unaffected by movement, but instead by the amount of force exerted onto the shell casing causing its explosion as it exits the gun's chamber, but hey, this is a book on *Quake III* programming, not ballistics! As you get into game development, you will discover that realism is important, but too much realism can often put a damper on the fun factor.

## Forward Is Such a Vague Term

When I talk about forward/backward movement, or left/right movement within the world of *Q3*, I do not always mean it literally. The velocity of the player is relative to the actual game world and how the player's movement relates to its x, y, and z coordinates (as mentioned earlier when I talked about the left-hand coordinate system.) Let me put it to you this way: If your player was placed in the game facing the direction of the positive y axis (towards the horizon), and you turned to the left 90 degrees and started walking forward, you would not be seen as moving "forward" to *Q3*. That's because you would actually be moving along the x axis (negatively), and your `ps.velocity[1]` would be the value changing, not `ps.velocity[0]`.

Keep this in mind when you are looking at player movement code. If the `velocity` variable is your only point of reference, it's far easier to determine that the player is simply moving as opposed to whether he is moving forward or backward (or to the left or right, for that matter). Luckily, these rules don't apply so much with the z axis. You can be assured that if the player's `ps.velocity[2]` value is positive, the player is moving straight up into the air. Likewise, a negative `ps.velocity[2]` indicates that the player is falling. I'll cover player movement and interaction in more detail in Chapter 4.

Make sure you also update your `cgame` code so that it acts in the same manner as your `game` code. To do so, pop open `cg_weapons.c`, scroll to the `CG_ShotgunPattern` function, and modify the `accuracyFactor` so that it equals 2, exactly as you did in `ShotgunPattern`:

```
vec3_t      end;  
vec3_t      forward, right, up;  
int         accuracyFactor = 2;
```

Then, rewrite the code that handles the checks for various player states. Because all of them must fire if (and only if) `otherEntNum` is equal to the client's `ps.clientNum` value, you can wrap all your accuracy

logic within that evaluation. Modify your code under the `CrossProduct` function so that it looks like so:

```
CrossProduct( forward, right, up );

if ( otherEntNum == cg.snap->ps.clientNum )
{
    // is the user crouching? bump up the accuracy!
    if (cg.snap->ps.pm_flags & PMF_DUCKED)
        accuracyFactor = 1;
    // is the user moving? lower the accuracy!
    else if (cg.snap->ps.velocity[0] || cg.snap->ps.velocity[1])
        accuracyFactor = 3;
    // is the user jumping? lower it more!
    else if (cg.snap->ps.velocity[2])
        accuracyFactor = 4;
}
```

Save your changes, compile your DLLs, drop them in your `MyMod` directory, and fire up *Q3*. Test it by adding a few bots to your game. I ran through `q3dm3` with three bots and found the changes quite interesting. Not only was I having a harder time shooting while running and jumping away from the bots, but they, too, were firing frequently without crouching. Thus, their accuracy also fell.

By taking the time to play with the shotgun, you have learned how to work with both the server-side game code as well as the client-side `cgame` code. You've also played with a few more variables in *Q3*, such as the player's `pmove` flags and `velocity`. With every change you make to *Q3*, you will come away with a better understanding of a few more elements in the game; ultimately, that will lead you to quicker development as you move to areas you haven't worked with yet.

## Modifying Grenades: The Cluster Grenade

For your next modification, I will show you how to modify the grenade's behavior in *Q3* so that it acts like a cluster grenade. That is, when the grenade explodes, it will break apart into three more grenades, which in turn will explode and cause extra damage.

Moving from the shotgun to the grenade launcher shouldn't be too difficult for you at this point. You already had a bit of experience playing with a missile-based weapon in the previous chapter, when you dealt with the rocket launcher. The grenade launcher is very similar in many respects: Both the rocket and grenade fire a missile-based object that is visible on the screen, and both missiles cause some hefty damage when they explode. Unlike the rocket, however, the grenade is affected by gravity, and has a bit of a bounce to it. Let's take a look at what defines this new behavior so you can prepare yourself better to modify the grenade's behavior.

## Further `g_weapon.c` Detective Work

Because the shotgun's base firing function ended up being `weapon_super_shotgun_fire`, let's take a look and see if we can find a similar function in `g_weapon.c` for the grenade. A quick search through the file reveals a function located around line 355, called `weapon_grenadelauncher_fire`:

```
void weapon_grenadelauncher_fire (gentity_t *ent) {
    gentity_t      *m;

    // extra vertical velocity
    forward[2] += 0.2f;
    VectorNormalize( forward );

    m = fire_grenade (ent, muzzle, forward);
    m->damage *= s_quadFactor;
    m->splashDamage *= s_quadFactor;
}
```

It seems to be a very simple function that starts by creating a `gentity_t` variable (which you should know by now is the type of entity on which all objects in *Q3* are based). It then gives the new entity a bit of a vertical boost by taking the global static variable `forward` (which represents the front of the player), and boosting its *z* axis by `0.2f`.

### TIP

The lowercase *f* at the end of `0.2f` forces the variable to be a 32-bit float; on its own, `0.2` would represent a 64-bit double. By explicitly expressing `0.2` as a 32-bit float, a slight speed increase in the calculation is gained.

That updated forward variable is passed to `VectorNormalize`, which rounds out its positions to save some bandwidth. Then, the new `gentity_t` is assigned to the value of the `fire_grenade` function, which you will look at in a moment. The `gentity_t`'s damage variable is multiplied by any active quad damage the player may have, as is its `splashDamage` variable.

So, if `fire_grenade` is the function that does the dirty work, let's find it. If you guessed that you would find the function in `g_missile.c`, give yourself another 50 bonus points. Pop open `g_missile.c` and scroll down to around line 562. Sure enough, there sits `fire_grenade`:

```
gentity_t *fire_grenade (gentity_t *self, vec3_t start, vec3_t dir)
```

Because a `gentity_t` is assigned to the result of `fire_grenade` back in `weapon_grenadelauncher_fire`, it comes as no surprise that this function returns a pointer to a variable of type `gentity_t`. It also takes a `gentity_t` as an input parameter, as well as two vectors: one for a starting location in the *Q3* world, and one to represent the direction it was fired.

## Why a Grenade Bounces (and Rockets Don't)

Although the core of the `fire_grenade` function is quite similar to that of the `fire_rocket` function from the previous chapter, there are a few key differences. Its `classname` property is not `rocket`, for starters; it's `grenade`. In addition, its `nextthink` value seems to be a lot lower, only the current time plus another 2.5 seconds (recall that the rocket's `nextthink` time was the current time plus an additional 15 seconds). There is also a line to modify the grenade's `entityState_t` property, which is held in the variable `s`:

```
bolt->s.eFlags = EF_BOUNCE_HALF;
```

### NOTE

Remember, missile weapons do two types of damage. The first type is *direct* damage, which is caused when the missile actually hits a physical target. *Splash* damage is the other type; a target within the vicinity of the missile's explosion will be affected by it. Typically, the closer a target is to the center of the explosion, the more damage is taken.

If you take a quick peek in `bg_public.h`, around line 230, you will see that `EF_BOUNCE_HALF` is one of many flags that can be assigned to an entity's state in `Q3`. This flag is especially important to the grenade, because it tells `Q3` that the grenade will bounce against surfaces that can't take damage when it makes contact with them (as opposed to instantly exploding like a rocket would). You should also note that the grenade's splash damage is somewhat higher than the rocket's (150 as opposed to 120), and of course, there are explicit flags that assign the grenade's "means-of-death" property. You see those in `MOD_GRENADE` and `MOD_GRENADE_SPLASH`.

The only other significant difference is the flag that is assigned to the grenade's position trace type, which is found in the `s.pos.trType` variable. For the rocket, it was a flag called `TR_LINEAR`, but in the case of the grenade, you can see that it is `TR_GRAVITY`. This is the flag that will tell `Q3` as each frame of animation is processed how to move the grenade throughout the world. By adding this flag, `Q3` understands that gravity will affect the grenade's trajectory as it is launched out of a weapon. To show you what kind of trace flags are available, I've added Table 3.2, which lists them in their entirety.

**Table 3.2** `trType_t` Flags

Variable	Value
<code>TR_STATIONARY</code>	This flag denotes an unmoving object.
<code>TR_INTERPOLATE</code>	This flag indicates an object whose motion can be "predicted" by <code>Q3</code> .
<code>TR_LINEAR</code>	This flag describes an object that moves in a constant straight line.
<code>TR_LINEAR_STOP</code>	This flag is the same as <code>TR_LINEAR</code> , except that it also indicates that the object has reached its endpoint.
<code>TR_SINE</code>	This flag describes an object that "bobs" up and down.
<code>TR_GRAVITY</code>	This flag is used on objects that are affected by gravity.



## Using What You Know: think and nextthink

You know two things coming into `fire_grenade` by having worked with the rocket's behavior code already. One: because the entity's `s.eType` flag is `ET_MISSILE`, you know that if the object hits a damageable target, the function `G_MissileImpact` will fire. Two: the grenade's `think` property is pointing to the `G_ExplodeMissile` function, just like the rocket's `think` property was. That means if the game's current time (held in `level.time`) ever exceeds the grenade's `nextthink` value, then `G_ExplodeMissile` will be the function that is called. To reiterate by using C-style comments:

```
bolt->s.eType = ET_MISSILE; // means G_MissileImpact will run if the
grenade hits a target
bolt->think = G_ExplodeMissile; // means G_ExplodeMissile will run if
time runs out. And time will run out when:
bolt->nextthink = level.time + 2500; // which means 2500 milliseconds
(or 2.5 seconds) from when the grenade is launched.
```

Now you have a crystal-clear picture of the functions that will run when the grenade explodes or comes into contact with a target. Let's start by writing a function that will control the explosion of the cluster grenade, so you can call it from both `G_MissileImpact` and `G_ExplodeMissile`.

Right above `G_ExplodeMissile` (which itself is above `G_MissileImpact`) is where you will define this new function. It's important to place the function in this spot, because you will not provide a function declaration or *prototype* for it. If you write a function that you do not declare, you must physically place it in your `.c` file *before any other function that will call it*. Above `G_ExplodeMissile`, add the following:

```
/*
=====
G_ExplodeCluster

Explode a cluster grenade into three shards
=====
*/
void G_ExplodeCluster( gentity_t *ent ) {
    vec3_t          dir;
```

```
VectorSet(dir, 33, 33, 10);
fire_cgrenade(ent->parent, ent->r.currentOrigin, dir);
VectorSet(dir, -33, 33, 10);
fire_cgrenade(ent->parent, ent->r.currentOrigin, dir);
VectorSet(dir, 0, -33, 10);
fire_cgrenade(ent->parent, ent->r.currentOrigin, dir);
}
```

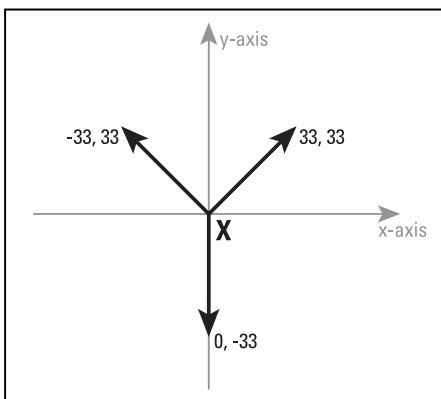
Here is the function that will launch three clusters after a grenade explodes. It is a very simple function that returns nothing, and requires one input parameter: a `gentity_t` that will be your original grenade. Within the function, you declare a `vec3_t` called `dir`, which will be used three times to set the direction of each cluster. You do this with the `VectorSet` function:

```
VectorSet(dir, 33, 33, 10);
```

`VectorSet` takes a `vec3_t` passed in the first parameter, and changes its `x`, `y`, and `z` positions by the amounts specified in the second, third, and fourth input parameters. You then take the updated `dir` variable and pass it to a new function, `fire_cgrenade`:

```
fire_cgrenade(ent->parent, ent->r.currentOrigin, dir);
```

You haven't written `fire_cgrenade` yet, but if you are keen observer, you will see that it takes the same input parameters as `fire_grenade`: a pointer to a `gentity_t`, a start vector, and a direction vector. By looking at the three `VectorSet` calls, you can guess that your three cluster grenades will break apart in triangular pattern, as noted by the different `x` and `y` values in each of the calls, similar to the diagram in Figure 3.5.



**Figure 3.5** The directions of the intended cluster grenades

Also, each cluster gets a bit of a kick upward by passing 10 into the final input parameter (which maps to the z axis). Now that you have `G_ExplodeCluster` in place, let's write `fire_cgrenade`.

## Making the Cluster Grenade Behave

The function `fire_grenade` is called outside of `g_missile.c` (you saw that it was the function that was called from `weapon_grenadelaucher_fire`), so it has been declared outside this file. You won't be calling `fire_cgrenade` from outside `g_missile.c`, but for sanity's sake (and so you don't have to worry about function placement within `g_missile.c`), let's declare it in `g_local.h`. Open `g_local.h` and scroll down to where `fire_grenade` is declared (around line 505). After `fire_grenade`'s declaration, add the following line:

```
gentity_t *fire_cgrenade (gentity_t *self, vec3_t start, vec3_t
aimdir); // new for clusters
```

Now let's add the function to `g_missile.c`. Start by copying everything in `fire_grenade` and pasting it after `fire_grenade`'s function completes, changing the name of the function after the paste. Also, change the velocity of the clusters so that they don't bounce as they would if fired directly from your grenade launcher. Then, find the call to `VectorScale` and change the 700 to 300. When you are finished, the code should look like the following:

```
/*
=====
fire_cgrenade
=====
*/
gentity_t *fire_cgrenade (gentity_t *self, vec3_t start, vec3_t dir) {
    gentity_t *bolt;

    VectorNormalize (dir);

    bolt = G_Spawn();
    bolt->classname = "grenade";
    bolt->nextthink = level.time + 2500;
    bolt->think = G_ExplodeMissile;
```

```
bolt->s.eType = ET_MISSILE;
bolt->r.svFlags = SVF_USE_CURRENT_ORIGIN;
bolt->s.weapon = WP_GRENADE_LAUNCHER;
bolt->s.eFlags = EF_BOUNCE_HALF;
bolt->r.ownerNum = self->s.number;
bolt->parent = self;
bolt->damage = 100;
bolt->splashDamage = 100;
bolt->splashRadius = 150;
bolt->methodOfDeath = MOD_GRENADE;
bolt->splashMethodOfDeath = MOD_GRENADE_SPLASH;
bolt->clipmask = MASK_SHOT;
bolt->target_ent = NULL;

bolt->s.pos.trType = TR_GRAVITY;
bolt->s.pos.trTime = level.time - MISSILE_PRESTEP_TIME; // move a
bit on the very first frame
VectorCopy( start, bolt->s.pos.trBase );
VectorScale( dir, 300, bolt->s.pos.trDelta );
SnapVector( bolt->s.pos.trDelta ); // save net bandwidth

VectorCopy (start, bolt->r.currentOrigin);

return bolt;
}
```

Now let's change the behavior of the original grenade. After all, it is the original grenade that will physically break up into the clusters. In order to do that, you must tell *Q3* that the grenade's entity name is something new. Scroll up to `fire_grenade` and change the `bolt->classname` to `cgrenade`. On the very next line, change the think time from 2500 (milliseconds) to 1500 (milliseconds). This will make the cluster break up faster than normal grenades took to explode on their own. After you make these changes, the first part of your `fire_grenade` function should look like this:

```
gentity_t *fire_grenade (gentity_t *self, vec3_t start, vec3_t dir) {
    gentity_t      *bolt;

    VectorNormalize (dir);
```

```

bolt = G_Spawn();
bolt->classname = "cgrenade"; // initially fire a cluster grenade
bolt->nextthink = level.time + 1500; // break apart faster than a
normal grenade explodes
bolt->think = G_ExplodeMissile;

```

Phew! You're almost there. Now all you need to do is make sure that when a cluster grenade is present in the world of *Q3* it is handled appropriately—that is, it breaks apart into three other grenades. This is why I quizzed you on `think` and `nextthink` earlier. You know that `G_MissileImpact` and `G_ExplodeMissile` are the functions that will handle a grenade's explosion, so let's go to those two functions and make your final adjustments.

`G_MissileImpact` is situated around line 264 in `g_missile.c`, so scroll up to it. Then, right near the end of the function, make the following change:

```

        if( !hitClient ) {
            g_entities[ent->r.ownerNum].client->accuracy_hits++;
        }
    }
}

// cluster grenades will spawn 3 new grenades on explosion
if (!strcmp(ent->classname,"cgrenade")) {
    G_ExplodeCluster( ent );
}

trap_LinkEntity( ent );

```

This is a very straightforward way to check the entity type—you use a function called `strcmp`, which stands for *string compare*. You compare the value of the string assigned to the `classname` property of the current object you're dealing with (`ent`), to the string `cgrenade`. If they match, you know you're dealing with a cluster grenade, and, thus, you need to fire the `G_ExplodeCluster` function you wrote earlier.

Let's go ahead and add the same string comparison on `ent->classname` to `G_ExplodeMissile`. This function is located way up at the top of `g_missile.c`, near line 56. As before, scroll to the end of the function

**NOTE**

If two strings are compared with the `strcmp` function, and they equal each other, `strcmp` will return 0. That may seem odd, because 0 typically evaluates to `false`—after all, you’re clearly looking to see if one string matches another, which sounds like that should be `true`. The reason for this relates to how the C programming language works. Testing for zero is *always* faster than testing for non-zero, so many functions that were built into C were made to work in this manner. That’s why you check for a `false` on the return of `strcmp`. Don’t be alarmed; many functions in C work this way.

and look for “cgrenade” in `ent->classname` as your last check, just before the call to `trap_LinkEntity`:

```
// splash damage
if ( ent->splashDamage ) {
    if( G_RadiusDamage( ent->r.currentOrigin, ent->parent, ent-
>splashDamage, ent->splashRadius, ent
    , ent->splashMethodOfDeath ) ) {
        g_entities[ent->r.ownerNum].client->accuracy_hits++;
    }
}

// cluster grenades blow themselves up differently
if (!strcmp(ent->classname, "cgrenade"))
    G_ExplodeCluster( ent );

trap_LinkEntity( ent );
```

Wow, that wasn’t so bad, eh? You don’t have to worry about any `cgame` modification this time around, because you’re not modifying the visual effects of the grenade. Instead, you’re modifying its behavior, which is fully controlled by the server portion of the code. Let’s compile and give it a run-through (see Figure 3.6).



**Figure 3.6** Your cluster grenade breaking apart

Hey, I think you're getting the hang of this weapon-modification bit! You could now take what you've learned with the cluster grenade and further refine its behavior, such as changing the velocity of the fragmented grenades (recall the 700 to 300 adjustment in `VectorScale` of `fire_cgrenade`), or even the number of grenades that fragment (`G_ExplodeCluster` calls `fire_cgrenade` three times, so why not four or five?). Don't be afraid to experiment with new ideas or changes. Any little modification you make may lead you to try a newer, better idea that may evolve into the next great weapon.

## A Further Adjustment: Gravity Wells

Because you're already working on the grenade, let's take a look at another possibility. Instead of having the grenade simply explode or break apart into more grenades, let's try something completely new: a gravity well. A *gravity well* is an object that pulls all other objects toward it. Wouldn't it be cool if you could have a grenade suck all nearby

opponents toward it like a vacuum? Then, when all the opponents near it get sucked too close, boom! You detonate the grenade on impact, damaging everyone that was caught in its gravitational field.

The gravity well is a really cool idea for a weapon mod, and one of its first uses was in a great deathmatch mod called *Painkeep*, for the original *Quake*. The creators of *Painkeep* went on to build their mod for *Q3*, and the gravity well made an encore appearance. In order to make grenades act like gravity wells, however, you must do something we haven't done before: Affect the physics of other players near the grenade. Is that possible? Absolutely! It's simply a matter of knowing what to work with.

## Into the Vortex

Start with a fresh code base so you don't mix your gravity-well code with the cluster-grenade code you just added. First, you need a way for your gravity well to find valid targets lying in its radius. The grenade is the center point in this case; you determine its range by granting it a given radius out from the center, generating a much larger circle in which its gravitational field will affect targets. How do you do this? Well, back when you modified the shotgun, you worked with a function called `trap_Trace`, which allowed you to draw a line toward a target to see if anything, such as an enemy, intersected that line. Optimally, you would like to use a similar function that looks within the area of a circle to see if any targets intersect the circle. Coincidentally, the programmers at id created a similar function, called `trap_EntitiesInBox` (shown here); you can re-use it in your gravity well's code.

```
int trap_EntitiesInBox( const vec3_t mins, const vec3_t maxs, int
*list, int maxcount )
```

Just like `trap_Trace`, `trap_EntitiesInBox` is a *system call* function in *Q3*, which means it has a direct line of communication into the *Q3* executable. This makes its performance extremely fast. It requires two vectors to be passed in, which represent the extents (or farthest points) of an invisible box in the world of *Q3*. Then, an array of integers is passed in; this is used by the function to hold any entities that positively pass the test for existence within the box's boundaries. Finally, an integer is passed in that represents the maximum amount of entities that will be in the array. (Because built-in arrays in C are



not self-describing, there is no way to extrapolate the length of the array by just the array itself.)

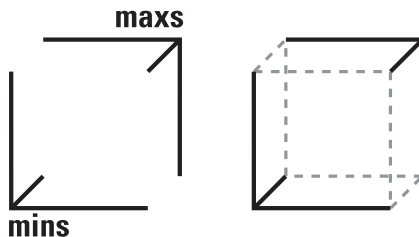
When all four of these parameters are passed into this function, a single integer is returned, which represents the number of all entities that were successfully found in the box.

## NOTE

Actually, there is one way of determining an array's length: by using the `sizeof` function. For all intents and purposes, however, it's often faster to define an upper limit of an array and simply use that instead. That's what ID did with `MAX_GENTITIES`, so I'm certain it's good enough for us.

### Building a Better Box with `mins` and `maxs`

When two `vec3_t` points are identified in 3D space, they can be used as farthest corners or *extents* of an invisible box. In Q3, these two points are often referred to as `mins` and `maxs`. By extending invisible lines toward each other, across each of the three axes (x, y, and z), these six lines eventually form the corners of a box that has an identifiable height, width, and depth, as shown here. This box can then be used in a number of calculations. For example, other `vec3_t` points can be tested to fall within the box. Or, the box itself can be used to represent the dimensions of a player, making it a *bounding box*. This assists in *collision detection*, the ability to tell when other objects pass into the box. You will be revisiting the concepts of `mins` and `maxs` throughout this book, and as well, be working with the player's bounding box in Chapter 4.



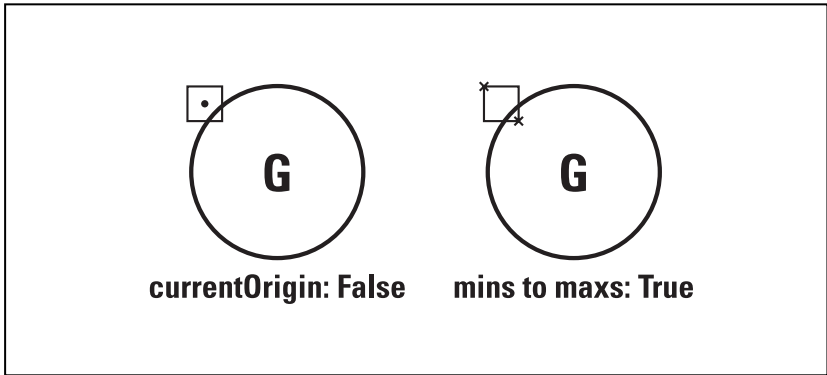
Knowing that, let's take a look at `trap_EntitiesInBox` in action. I've taken the liberty of writing you a new think function for the grenade, based off existing code created and refined by Karl Pauls. This update will make the grenade behave like a gravity well. Take a look at this function now; it is called `G_Vortex`, and appears in the next code listing, which I have numbered, so that we can refer back to specific parts.

```
1.  /*
2.  =====
3.  G_Vortex
4.  =====
5.  */
6.  #define GVORTEX_TIMING          10          // the think time interval
   of G_Vortex
7.  #define GVORTEX_VELOCITY       10000       // the amount of kick each
   second gets
8.  #define GVORTEX_RADIUS         500         // the radius of the gravity
   well
9.
10. static void G_Vortex ( gentity_t *self )
11. {
12.     qboolean    explode = qfalse;
13.     float       dist;
14.     gentity_t   *target;
15.     vec3_t      start, dir, end, kvel, mins, maxs, v;
16.     int         entityList[MAX_GENTITIES], numListedEntities, i,
   j;
17.
18.     target = NULL;
19.
20.     for ( i = 0 ; i < 3 ; i++ ) {
21.         mins[i] = self->r.currentOrigin[i] - GVORTEX_RADIUS;
22.         maxs[i] = self->r.currentOrigin[i] + GVORTEX_RADIUS;
23.     }
24.
25.     numListedEntities = trap_EntitiesInBox( mins, maxs,
   entityList, MAX_GENTITIES );
26.
27.     for(j = 0 ; j < numListedEntities ; j++) {
```

```
28.
29.         target = &g_entities[entityList[ j ]];
30.
31.         for ( i = 0 ; i < 3 ; i++ ) {
32.             if ( self->r.currentOrigin[i] < target->r.absmin[i] )
33.                 {
34.                     v[i] = target->r.absmin[i] - self-
35.                         >r.currentOrigin[i];
36.                 } else if ( self->r.currentOrigin[i] >
37.                     target->r.absmax[i] ) {
38.                     v[i] = self->r.currentOrigin[i] -
39.                         target->r.absmax[i];
40.                 } else {
41.                     v[i] = 0;
42.                 }
43.         }
44.
45.         dist = VectorLength( v );
46.
47.         if ( dist > GVORTEX_RADIUS )
48.             continue;
49.
50.         if (target == self)
51.             continue;
52.
53.         if (!target->client)
54.             continue;
55.
56.         if (target == self->parent)
57.             continue;
58.
59.         if (!target->takedamage)
60.             continue;
61.
62.         if (target->health < 1)
63.             continue;
64.
65.         explode = (dist <= 5) ? qtrue : qfalse;
```

```
63.     VectorCopy(target->r.currentOrigin, start);
64.     VectorCopy(self->r.currentOrigin, end);
65.     VectorSubtract(end, start, dir);
66.     VectorNormalize(dir);
67.     VectorScale(dir, GVORTEX_VELOCITY / GVORTEX_TIMING, kvel);
68.     VectorAdd(target->client->ps.velocity, kvel,
        target->client->ps.velocity);
69.
70.     if ( !target->client->ps.pm_time ) {
71.         target->client->ps.pm_time = GVORTEX_TIMING - 1;
72.         target->client->ps.pm_flags |= PMF_TIME_KNOCKBACK;
73.     }
74.
75.     VectorCopy(dir, target->movedir);
76.
77.     if (explode)
78.         G_ExplodeMissile( self );
79. }
80.
81. self->nextthink = level.time + GVORTEX_TIMING;
82. if (level.time > self->wait)
83.     G_ExplodeMissile( self );
84. }
```

Wow, there is a lot going on in this function, so let's dissect it a little bit at a time so that everything is clear. First, you start by setting a few global variables (line 6 through 8) via the `define` keyword. `GVORTEX_TIMING` represents the time your gravity well updates itself (in milliseconds), rechecking for new targets, while `GVORTEX_VELOCITY` is a value used to represent how hard you will suck your targets toward the grenade. Finally, you set a variable called `GVORTEX_RADIUS`, which is used to determine how wide an area your gravity well will affect. It generates a vector that represents the absolute extents of the target's bounding box, including any degree of rotation involved, which acts as a more realistic way to test for collision. You could have made a call to the `Distance` function, which takes two origins as parameters (the origin of the grenade and the origin of the player), but a center point of a target may not fall within a radius, whereas its outer extents may (see Figure 3.7). Doing this extra bit of work just makes for a more realistic test.



**Figure 3.7** The center of the target does not fall within the grenade’s area of affect (left), but the extents of its `mins` and `maxs` value create a box that does.

When the function begins (line 10), start by creating your own temporary extents, which will make up the area of affect for the gravity well. Do this by creating two vectors, a `mins` and a `maxs`, and set them to the origin of grenade minus the radius, and plus the radius, respectively. A quick refresher of simple geometry reveals that a circle is as wide as its radius \* 2, so by taking a center point, and by both adding and subtracting a radius, you successfully create the circle’s *diameter*, or the width of the circle.

Once you have this “area of effect” circle, you can begin testing for existence of targets by calling `trap_EntitiesInBox` (line 25), passing in your newly created extents (`mins` and `maxs`), as well as your array of integers to hold the found entities and your entity upper limit (`MAX_GENTITIES`). You assign the results of this function

call to an integer named `numListedEntities`. Next, begin looping over this number so that you can test each entity for validity. Each entity to test is set by assigning a temporary `gentity_t` variable, `target` (line 29),

### TIP

Technically, you don’t create a circle when you use `trap_EntitiesInBox` because a `mins` and a `maxs` value create a three-dimensional cube. By using the cheat of adding and subtracting a radius (line 21 & 22), however, you can fake the creation of the circle, achieving a relatively good approximation (which would be a sphere in 3D space).

to the value of the matching entity in the global game variable `g_entities`. Every single entity in game will be found in the `g_entities` array somewhere, so this is the way you handle entity lookup.

## Testing for Collision

The next thing you need to take care of is determining whether there is a valid connection between the entity's bounding box and your grenade's area of affect. Do so by borrowing a bit of code that ID used in another function (the `for` loop over all three axes on line 27). This snippet of code will give you a vector equal to the distances found from the target to the grenade. Pass this vector to `VectorLength`, which returns a distance you can begin to test with. Then, do some sanity checking on your distance and target information (on line 43):

1. If the distance between the grenade and the target is greater than the gravity well's radius, skip it.
2. If the target is equal to the gravity well, skip it.
3. If the target is not a client (an active player), skip it.
4. If the target is the client that fired the gravity well, skip it. The attacker, in this case, is ignored by the gravity well's field.
5. If the target cannot take damage, skip it.
6. If the target is dead, skip it.

If all those checks pass, then you can do your dirty work. Start by assigning a value to your `explode` variable, which will either be `true` or `false`. You want the grenade to explode if a target is pulled within 5 units of the grenade, so if your newly created distance variable is less than or equal to 5, you set that `explode` variable to `true`; otherwise, it's set to `false` (line 61).

Next (starting at line 63), assign a `start` vector equal to the target's position, and assign an `end` vector equal to the gravity well's position. Then, subtract these two vectors to get a `dir`, or direction vector (the direction the target will move toward the gravity well). Normalize the three coordinates in the vector to get a fixed distance, and then scale the direction vector by a calculation of the gravity well's velocity divided by the `think` time. Place this result, `kvel`, into your target's velocity (line 68). Finally, set the target's move direction equal to the new direction you have created (line 75).

**NOTE**

There is an additional bit of code used here (at line 70) that deals with the `PMF_TIME_KNOCKBACK` flag. This code is placed here so that if the velocity of the target is somehow acted upon by another object (say, another gravity well), the target's velocity is not suddenly negated, which would cause a jerky movement. This same code is used by `id` when `G_Damage` is used to move a target with knock back, and in the `TeleportPlayer` function, which kicks a player out of a new teleport destination.

Wow, did you make it through all that? If so, congratulate yourself! Working with vectors, velocity, and move directions definitely allows you to chalk up some experience in the physics department.

Next, check to see if the `explode` flag was set to `true` (line 77), and if so, call `G_ExplodeMissile` on the gravity well. Otherwise, the loop continues through the rest of the found targets, applying the same logic to each of them in succession.

The function begins to wrap up by adding your `GVORTEX_TIMING` value to its `nextthink` property (line 81), making it perform this entire function call again 0.1 seconds in the future. `G_Vortex` concludes with a check to see whether the current game time is greater than the gravity well's `wait` property, and if so, it calls `G_ExplodeMissile` to finish off the life span of the grenade.

## Making Gravity Work for You

The last thing you need to do is modify the grenade's initial behavior when launched, so that it calls `G_Vortex` and not `G_ExplodeMissile` when its `nextthink` time is up. Scroll to around line 577 in `g_missile.c` and make the following changes:

```
bolt = G_Spawn();  
bolt->classname = "grenade";
```

```

    bolt->nextthink = level.time + 1000; // G_Vortex will run in 1
second.
    bolt->think = G_Vortex;           // G_Vortex is our dirty
doer!
    bolt->wait = level.time + 20000;   // Wait for 20 seconds before
any final explosion
    bolt->s.eType = ET_MISSILE;

```

By changing the grenade's think function to `G_Vortex`, you change its behavior as it enters the *Q3* world, because that will be the function that is called when the current game time equals or exceeds the grenade's `nextthink` property. You have also modified `nextthink` to be only 1000 milliseconds from launch time, so it starts calling `G_Vortex` within 1 second of being fired. Furthermore, the `wait` property of the grenade is modified, setting it to the current time, plus an additional 20 seconds. Because `G_Vortex` is the only function that is called by the grenade while it's in the game world, the `wait` property allows you to call an additional function when a certain amount of time has elapsed, which you'll recall ends up being `G_ExplodeMissile`.

That's it! You're ready to compile and give it a run. I quickly tried it with a few bots on *q3dm3* and was excited to see the bots unknowingly get sucked toward the grenade and blown up on impact. You can go back and tweak the `GVORTEX_TIMING`, `GVORTEX_VELOCITY` and `GVORTEX_RADIUS` variables to create different styles of the gravity well. With the current defaults, the gravity well is quite powerful—it draws targets toward it with insane speed. Lowering the `GVORTEX_VELOCITY` variable will create a slower pull toward the grenade.

## Summary

You've done some considerable work up to this point. For example, you've gotten to know the different types of weapons that are in *Q3* and what makes them tick. You've also had a chance to modify different types of behaviors, from aim accuracies to making the weapons affect the physics of the players around them. And, while working with all this weapon code, you've begun to see how different pieces of information in *Q3* can be extrapolated, such as what the player is doing, or how much time has elapsed since a certain event. I'd wager



that you have a pretty good understanding of vectors by now, as well. You will revisit weapon code one more time before this book ends, when you take a deeper look into the `cgame` code and how to create weapons that use new client effects. For now, though, because the code of dealing with player's physics is still fresh in your mind, let's turn to working directly with the *Q3* player itself.



# CHAPTER 4

# MANIPULATING THE PLAYER

**M**odifying how the weapons in *Q3* behave is reasonably simple, primarily involving the changing of properties, such as the speed of a weapon's projectile or how much damage a gun's blast causes. When you manipulate the player and his actions, however, things get a lot more complex—and a lot more fun. You've gotten a taste of this already, through the gravity-well mod in Chapter 3. Bouncing the player around by playing with vectors and velocity, however, is just the start. In this chapter, you'll delve into how the player interacts with his world and how he maintains his information from fight to fight. You'll also learn some new techniques for investigating the code yourself, gaining knowledge about solving some of these problems on your own. These techniques will be essential for you after you complete this book.

## The *Quake III* Player and His World

There would be no *Quake* without players. It makes sense, then, that the player, an integral part of *Q3*, is dealt with in explicit detail. On the surface, the player is a fairly straightforward character that you might find in any FPS. He has a set of statistics, such as health and armor, and a list of weapons currently being held with ammo for each one. As for his physical presence, he is rendered in the 3D world of *Q3* by a *model*, a series of polygons placed together to create the form of a human being (or alien, if that happens to be your model of choice). The model is *skinned*, which means it has lifelike images or textures applied to the surfaces of the polygons. This creates

### NOTE

A *polygon* is simply a closed-plane figure that is bounded by straight lines. The simplest polygon is the *triangle*, as it meets the minimum requirement of having three sides. All polygons can be broken down into triangles.

the effect of a more believable character. When you see the player's model in the game, you can easily associate it in your mind with an actual *Q3* game player. And of course, this character model has animations. He walks, runs, crouches, jumps, and fires a multitude of weapons; he even has the ability to taunt his opponent with a crude gesture.

To a programmer, however, the player is a complicated and extremely detailed piece of code. The player knows how it is killed, and by whom, and with what weapon. The player has its own internal score-keeping system, such as how many frags it has accrued, how many times it has died, and so on. In team games like *Capture the Flag*, the player even knows how many times it has captured, or has assisted in a capture, or has successfully defended the flag from being taken. There is a significant amount of code dedicated simply to initializing the player's 3D model within the world of *Q3*.

The code behind the player even has the ability to segregate and capture certain variables to be held in different states, which are uniquely organized into one of three groups:

- **Variables to be held across sessions, between full-level loads, as long as the player persists on the server.** Examples of these are whether the player is a spectator, how many wins and losses the player has, and in team games, whether the player has been designated a team leader.
- **Variables to be held across multiple respawns (such as after the player is killed), but not between level loads.** Variables that fall into this category include the player's name (as other players see it in the game and on the scoreboard), the time that the player entered the level, and how many times the player has called a vote.
- **Variables that are held only between client spawns.** Every time a player dies and respawns in the map, these variables are reset. You've already worked with some of these variables, such as the direction of the player's movement, whether the player is crouching (remember `pm_flags?`), the player's velocity, and the currently selected weapon.

This is just scratching the surface of the player's code; there is much more to be learned about what the player can see and do from within

Q3. To get a better understanding of just what controls all this data, let's take a look at the structs that define what a player is.

## Player Structure

Ironically, the player's structure is made up of several important structs in Q3. To refresh your memory, a *struct* is a variable that is defined within C to contain other types of variables in a tree-like hierarchy that can be referenced using dot notation (shown in a moment). Let me give you an example:

```
typedef struct {
    int      health;
    char     *name;
    float    dir;
} player_t;
```

Here, a simple struct is defined, which itself contains an integer called *health*, a char pointer called *name*, and a float called *dir*, which will hold a decimal value equal to a compass direction between 0 and 360. The newly created struct is called *player\_t*. This code snippet in and of itself is not enough to be used in a program; another internal variable must be declared to be of type *player\_t*. For example:

```
player_t    myClient;
```

Magic! Now you have a newly defined variable, called *myClient*, for use in your program, and it is declared as a variable of type *player\_t*, which is your struct. Therefore, using the previously mentioned dot notation, you should be able to read and write to the values of *player\_t*'s properties or *members*, such as:

```
myClient.health = 300;
myClient.name = "Learless|M";
myClient.dir = 180.7;
```

With this code, the new variable's *health* is set to 300, the *name* to "Learless|M", and the *dir* to 180.7.

Because there are no limitations as to what types of variables can be used to define members of a struct, a struct can also contain other structs, which can result in some fairly complicated code snippets. As

## Being Directly Indirect

There is one time when you do not access the properties of a struct with dot notation: when you pass a pointer to a struct into a function. Because structs can be large, complex variable types, it is often much more efficient to pass a pointer to a struct into a function (and this is done frequently in *Q3*). When dealing with a struct that has been passed into a function via a pointer, you access its properties indirectly, using *pointer notation*, or the `->` symbol. So, if the `myClient` variable had been passed to a function as a pointer, and you wanted to set the value of the `health` property to 0, you would do it with the following syntax:

```
myClient->health = 0;
```

you feel the need to create and manage new structs, strive to keep their complexity to a minimum; structs of structs can easily get out of hand and will only serve to confuse you more.

## The Guts of `gclient_s`

I mentioned in Chapter 3 that everything in *Q3* is, ultimately, an entity. The object representing the player is no different. Beginning at the top of the *Q3* code hierarchy, in the definition of the struct that creates all entities (`gentity_s`), you can see a reference to a variable called `client`. Open up `g_local.h` and scroll to line 58:

```
struct gclient_s    *client;                // NULL if not a client
```

Here, a variable named `client` is defined, which points to a data type called `gclient_s`. Since the `gentity_s` struct is the basis for every entity in *Q3*, every full-fledged player in *Q3* will have a pointer to the `client` variable, absorbing all the data that is contained within a `gclient_s` struct.

### Saving Keystrokes with typedef

In Q3, the structs `gclient_s` and `gentity_s`, which are the basis for all clients and entities, respectively, have been also named `gclient_t` and `gentity_t`. This is done via the keyword `typedef`, occurring on line 47 of `g_local.h`. The `typedef` keyword is used in C to create new names for existing types. Since structs are declared via the `struct` keyword, any time a variable that is of type `gclient_s` or `gentity_s` needs to be referenced in the code, it must be prefaced with the `struct` keyword. To save this extra typing, a `typedef` of each struct was created, so it could be referenced by a single word. The naming convention that id Software decided on was `_s`, to refer to the actual struct, and `_t`, to refer to the `typedef`. You will see this naming convention throughout the Q3 code base (and throughout this book).

So remember, whenever you see:

```
gentity_t
```

you are really seeing:

```
struct gentity_s
```

So, then, the next task is to determine what the makeup of a `gclient_s` struct is. Jump down to line 241 in the same file, and you should see the following code:

```
struct gclient_s {
    // ps MUST be the first element, because the server expects it
    playerState_t      ps;          // communicated by server to clients

    // the rest of the structure is private to game
    clientPersistant_t pers;
    clientSession_t   sess;

    qboolean    readyToExit;        // wishes to leave the intermission
```

```
qboolean    noclip;

int         lastCmdTime;        // level.time of last usercmd_t,
for EF_CONNECTION

// we can't just use
pers.lastCommand.time, because
// of the g_synchronousclients case

int         buttons;
int         oldbuttons;
int         latched_buttons;

vec3_t      oldOrigin;

// sum up damage over an entire frame, so
// shotgun blasts give a single big kick
int         damage_armor;      // damage absorbed by armor
int         damage_blood;      // damage taken out of health
int         damage_knockback;  // impact damage
vec3_t      damage_from;       // origin for vector calculation
qboolean    damage_fromWorld;  // if true, don't use the dam-
age_from vector

int         accurateCount;     // for "impressive" reward sound

int         accuracy_shots     // total number of shots
int         accuracy_hits;     // total number of hits

//
int         lastkilled_client; // last client that this client
killed
int         lasthurt_client;   // last client that damaged this
client
int         lasthurt_mod;      // type of damage the client did

// timers
int         respawnTime;       // can respawn when time > this,
force after g_forcerespwan
int         inactivityTime     // kick players when time > this
qboolean    inactivityWarning  // qtrue if the five second warn-
ing has been given
```



```

    int          rewardTime;          // clear the EF_AWARD_IMPRESSIVE,
etc when time > this

    int          airOutTime;

    int          lastKillTime;        // for multiple kill rewards

    qboolean     fireHeld;            // used for hook
    gentity_t    *hook;              // grapple hook if out

    int          switchTeamTime      // time the player switched teams

    // timeResidual is used to handle events that happen every second
    // like health / armor countdowns and regeneration
    int          timeResidual;

#ifdef MISSIONPACK
    gentity_t    *persistantPowerup;
    int          portalID;
    int          ammoTimes[WP_NUM_WEAPONS];
    int          invulnerabilityTime;
#endif

    char         *areabits;
};

```

Wow, that is a one giant struct! The first three variables may ring a bell; they refer to the three states that a player maintains throughout the course of playing *Q3*. The first, *playerState\_t*, is a struct that maintains values only between respawns. If your player dies in battle, you can expect that everything in the *playerState\_t* struct will reset.

The second struct, called *clientPersistant\_t*, is responsible for carrying player information for the duration of an entire level. If you were creating a mod like *Q3 Fortress*, where players are given the choice of selecting a specific class to play, you would want to hold that class in the *clientPersistant\_t* struct so that if the player dies, his class is remembered.

The last of the big three structs is *clientSession\_t*, which maintains specific player information as long as the player persists on the server. That includes keeping variables after a player dies and respawns, and

## Preprocessor Directives

---

There is a strange bit of code in the previous listing which you may have seen already in your travels through the Q3 source. The following line,

```
#ifdef MISSIONPACK
```

is called a *preprocessor directive*. That means it is a special bit of logic that does not get compiled into the code, but rather, is interpreted by the compiler during compilation to specify certain parameters you would want only on a given build. This particular directive says “If the current build being compiled is a *Quake III* Team Arena Mission Pack build, include the following lines of code; otherwise, skip them.”

The reason for this is that the mission pack for Q3 contains many new references to models, powerups, rules, and game types that don’t normally exist in standard Q3. All that additional stuff requires additional code. If you unnecessarily include the extra mission-pack code in your standard Q3 builds, and then run your mod (without the mission pack being installed), Q3 will throw all kinds of crazy errors. This particular preprocessor directive ensures that mission-pack builds get the necessary code, while standard Q3 builds do not.

even after levels change. As I mentioned earlier, a good value to hold here is whether the player is spectating the current battle, because he would most likely want to remain spectating if a level changes.

The remainder of variables found in `gclient_s`, such as `damage_armor`, `accuracy_hits`, and `lastkilled_client`, are all treated as `playerState_t`’s variables; that is, they expire when the player expires. `gclient_s` has quite a number of other variables as its members; most of them are reasonably well documented right in the source, so feel free to take a look at the remainder of the struct. Your focus for the duration of this chapter is going to be these three structs. Let’s start by taking a look at what values are held only between player respawns.

## State of the Player

In order to break down the mystery of the `playerState_t` struct, you need to go beyond `g_local.h` and look in a file called `q_shared.h`. This file has a little more meat to it, because every other file in the *Q3* source references it. For exactly that reason, changes to this file should not be made lightly; ID itself makes that point very clear in several detailed C comments throughout the file. The struct you are looking for is found deep within this file, around line 1114:

```
typedef struct playerState_s {
    int          commandTime;      // cmd->serverTime of last executed
command
    int          pm_type;
    int          bobCycle;        // for view bobbing and footstep
generation
    int          pm_flags;        // ducked, jump_held, etc
    int          pm_time;

    vec3_t      origin;
    vec3_t      velocity;
    int         weaponTime;
    int         gravity;
    int         speed;
    int         delta_angles[3];  // add to command angles to get view
direction
                                // changed by spawns, rotating
objects, and teleporters

    int         groundEntityNum;  // ENTITYNUM_NONE = in air

    int         legsTimer;        // don't change low priority
animations until this runs out
    int         legsAnim;        // mask off ANIM_TOGGLEBIT

    int         torsoTimer;      // don't change low priority
animations until this runs out
    int         torsoAnim;      // mask off ANIM_TOGGLEBIT

    int         movementDir;     // a number 0 to 7 that represents
the reletive angle
```

```

// of movement to the view angle
(axial and diagonals)
// when at rest, the value will
remain unchanged
// used to twist the legs during
strafing

    vec3_t      grapplePoint;    // location of grapple to pull
towards if PMF_GRAPPLE_PULL

    int         eFlags;          // copied to entityState_t->eFlags

    int         eventSequence;   // pmove generated events
    int         events[MAX_PS_EVENTS];
    int         eventParms[MAX_PS_EVENTS];
    int         externalEvent;   // events set on player from another
source
    int         externalEventParm;
    int         externalEventTime;

    int         clientNum;       // ranges from 0 to MAX_CLIENTS-1
    int         weapon;          // copied to entityState_t->weapon
    int         weaponstate;

    vec3_t      viewangles;      // for fixed views
    int         viewheight;

    // damage feedback
    int         damageEvent;     // when it changes, latch the other
parms
    int         damageYaw;
    int         damagePitch;
    int         damageCount;

    int         stats[MAX_STATS];
    int         persistant[MAX_PERSISTANT]; // stats that aren't
cleared on death
    int         powerups[MAX_POWERUPS];    // level.time that the
powerup runs out
    int         ammo[MAX_WEAPONS];
```

```

int         generic1;
int         loopSound;
int         jumppad_ent;          // jumppad entity hit this frame

// not communicated over the net at all
int         ping;                // server to game info for score-
board
int         pmove_framecount;    // FIXME: don't transmit over the
network
int         jumppad_frame;
int         entityEventSequence;
} playerState_t;

```

It is evident that there is quite a bit of data to hold between respawns. Some items may look familiar, such as the `pm_flags` integer, the velocity vector, and the integer that represents `movementDir`. You may also remember `eFlags`, `clientNum`, and `weapon`. You will be playing with some more of these variables in this chapter.

The next struct you will want to look at is `clientPersistant_t`, which you looked at briefly in Chapter 2. It's shown next.

```

typedef struct {
    clientConnected_t    connected;
    usercmd_t            cmd;          // we would lose angles if not
persistent
    qboolean             localClient;  // true if "ip" info key is
"localhost"
    qboolean             initialSpawn; // the first spawn should be at a
cool location
    qboolean             predictItemPickup; // based on cg_predictItems userinfo
    qboolean             pmoveFixed;    //
    char                 netname[MAX_NETNAME];
    int                  maxHealth;     // for handicapping
    int                  enterTime;     // level.time the client entered
the game
    playerTeamState_t    teamState;     // status in teamplay games
    int                  voteCount;     // to prevent people from
constantly calling votes
    int                  teamVoteCount; // to prevent people from
constantly calling votes
    qboolean             teamInfo;     // send team overlay updates?
} clientPersistant_t;

```

You'll recall that in Chapter 2, you added the `homing_status` variable to this struct, which acted as a toggle (on/off) switch to determine what type of missile the player had selected. Because you placed in it this struct, you systematically made this variable's state persist for the duration of the entire level of play. When the player died, and then respawned in the game, his previous selection of `homing_status` was retained.

Finally, the third state of a player variable can reside in the struct `clientSession_t`, which is shown below, and can be found on line 206 of `g_local.h`.

```
typedef struct {
    team_t      sessionTeam;
    int         spectatorTime;    // for determining next-in-line to
play
    spectatorState_t  spectatorState;
    int         spectatorClient;  // for chasecam and follow mode
    int         wins, losses;     // tournament stats
    qboolean    teamLeader;      // true when this client is a team
leader
} clientSession_t;
```

Any variable set in this struct will remain intact as long as the player remains on the server—that includes between deaths and between level changes. Unlike with the previous members of the `gclient_s` struct, which represents all the player's data, it is not enough to simply add a new variable to this struct and assume its state will be maintained automatically. The `clientSession_t` struct's members are actually maintained by reading and writing to a console variable, or *Cvar*, during level changes.

## Changing the Player's Movement

Now that you have a better understanding of the player's internal variable construct, let's take a look at what you can do to modify it. The easiest place to start modification is with the player's movement. In order to change any of the player's movement variables, you must visit the player's `think` function that is called every frame of animation. That function is `ClientThink_real`, and can be found on line 736 of `g_active.c`.

## Playing with ps.speed

ClientThink\_real is a think function, not unlike the ones you have dealt with in previous chapters. It is called by *Q3* about once for every frame of animation within the current game. This is where many playerState\_t values are read, set, or updated, due to changes in the game. One of these members is speed. A good example of how the player's speed is modified in the *Q3* code is via the haste rune, a powerup in *Q3* that allows the user to speed around the map much faster than normal.

If you open g\_active.c and jump to line 831, you'll see the following tidbit of code.

```
if ( client->ps.powerups[PW_HASTE] ) {
    client->ps.speed *= 1.3;
}
```

This simple piece of logic looks at the client variable, which you'll remember is of type gclient\_s, the gigantic struct of variables that control the player. It is specifically accessing a member of playerState\_t, as represented by the ps variable, which is being pointed to by the client variable (remember, members of structs are accessed via dot notation or pointer notation, the . or -> symbols, respectively). These few lines of code read in plain English, as follows: "If the player currently has the haste powerup, multiply the player's current speed by 1.3."

### NOTE

The strange \*= notation used in the haste example is a common bit of shorthand in the C language for variable assignment. Generally, you might see something like this:

```
variable = variable * 3;
```

This sets the value of a variable equal to itself, multiplied by 3. You can shorten that to the following:

```
variable *= 3;
```

This type of assignment can also be used with the +, - and / operators.

The player's speed property is very easy to change. For example, you could double the effects of the haste rune by changing the line to

```
client->ps.speed *= 2.6;
```

Or, you could create a new powerup that slows the player down (and possibly does something else, like double his melee damage) by halving its current value like so:

```
client->ps.speed /= 2;
```

Speed modification is the simplest of procedures, so for now, keep that nugget of information in the back of your head. You'll use it later on. Now let's take a look at more facets of player movement.

## Gravity Kills

Another `playerState_t` member that is updated in `ClientThink_real` is `gravity`. Gravity, as you know, is the "pulling" effect that large mass objects have on smaller objects. In the world of *Q3*, gravity must also be simulated in a reasonable fashion. That means that any object that is affected by gravity and not standing on a surface must have downward movement applied to it.

Because you have already worked with the player's movement in the world of 3D, you should know that in order to move any object downward, you must apply a negative vector to the player's z axis. Thankfully, the *Q3* engine already handles this for you, so you need not worry about applying any kind of physics algorithms to the player to force him downward. All you need to worry about is the amount of gravity currently in use in the game, and how much of it is applied to the player.

### NOTE

The acceleration an object undergoes due to gravity is a constant value, often referenced in math and physics texts as *g*. Its value is 9.81 (meters/sec) per second.

By default, a *Q3* game has an amount of gravity equal to 800. This value is simply an arbitrary number used by the engine to determine how to properly calculate the acceleration of objects moving



downward. It is applied to the player every frame in `ClientThink_real`, on line 820 of `g_active.c`:

```
client->ps.gravity = g_gravity.value;
```

Once again, you can see that the `playerState_t` struct's `gravity` member is being set to a value—in this case, the value of another struct's member, that of `g_gravity.value`. `g_gravity` maps to a `Cvar` that can be set and controlled by the server. You could take this line of code and adjust the gravity being applied to the player based on a powerup. For example, if the user had the `Invisibility` powerup, you might drop the gravity for the user slightly:

```
if ( client->ps.powerups[PW_INVIS] ) {
    client->ps.gravity = g_gravity.vaue / 1.4;
}
```

## NOTE

The `powerups` member of `playerState_t` is an array. Each index of the array is an enum of type `powerup_t`, which you can see declared in `bg_public.h` on line 247. I'll cover the enum data type in more detail in Chapter 6, "Client Programming."

Another idea to try might be to increase the effects of gravity on a player that has the `BFG` in his inventory of weapons:

```
if ( client->ps.stats[STAT_WEAPONS] & (1 << WP_BFG) )
    client->ps.gravity = g_gravity.value * 2;
```

Again, you can see that this is a fairly simple stat to modify.

## The Case of the Missing client

After having dealt with player speed and gravity, you can look to a third movement option: that of how high the player can jump. The player's jump velocity is a hard-coded value, set near the top of `bg_local.h`:

```
#define JUMP_VELOCITY 270
```

By *hard-coded*, I simply mean that there is no other state in the game that changes this variable. It is not Cvar that can be changed on the server, nor is it a value that can change based on powerups, style of game, health, weapons, or any other event you might think of. Every single player gets the same jump velocity. Well, that's no fun, so let's see what you can do to change it.

The most obvious change would be to modify `#define` at the top of `bg_local.h`, but that doesn't solve the problem of the variable being *constant*. Once a constant variable is declared in C, it cannot change throughout the entire course of the program's execution. You want to find a way to dynamically change the player's jump velocity based on some kind of criteria, such as a powerup or amount of health. The player's movement code is mostly handled in a file called `bg_pmove.c`, so let's open that file now.

Around line 339, you'll come to a function called `PM_CheckJump`, which is called when the player attempts to jump within `Q3`. A bit more sleuthing reveals that on line 361, the magic happens:

```
pm->ps->velocity[2] = JUMP_VELOCITY;
```

From this, you can extrapolate that the z axis (held in the third index) of `velocity` in the `playerState_t` struct is set to the value of the `JUMP_VELOCITY` variable as defined at the top of `bg_local.h`. Sounds simple enough; setting the z axis's velocity of an object equal to a positive value would result in that object moving upward, hence a jump. Except there is one small problem: where the heck is the `client` variable? To unravel this mystery, you work backward through the functions you know to figure out how to arrive at `pm->ps->velocity`.

## Solving the Jumping Mystery

Let's take a look at `ClientThink_real`, where you're sure to extract valid player data from the `client` variable. You've already played with gravity and speed in this function, so you are guaranteed access to the `client` variable. As the function begins, a number of variables are declared, including `pm`, which is of type `pmove_t`, and a pointer called `ucmd`, which points to a variable of type `usercmd_t`. These are two new structs that you have not dealt with yet, so let's take a look at them.

The `pmove_t` struct is defined at the top of `bg_public.h`, at line 140.

```
typedef struct {
    // state (in / out)
    playerState_t    *ps;

    // command (in)
    usercmd_t        cmd;
    int               tracemask;           // collide against these
types of surfaces
    int               debugLevel;         // if set, diagnostic output
will be printed
    qboolean          noFootsteps;       // if the game is setup for
no footsteps by the server
    qboolean          gauntletHit;       // true if a gauntlet attack
would actually hit something

    int               framecount;

    // results (out)
    int               numtouch;
    int               touchents[MAXTOUCH];

    vec3_t            mins, maxs;        // bounding box size

    int               watertype;
    int               waterlevel;

    float             xyspeed;

    // for fixed msec Pmove
    int               pmove_fixed;
    int               pmove_msec;

    // callbacks to test the world
    // these will be different functions during game and cgame
    void              (*trace)( trace_t *results, const vec3_t start,
const vec3_t mins, const vec3_t maxs, const vec3_t end, int
passEntityNum, int contentMask );
    int               (*pointcontents)( const vec3_t point, int
passEntityNum );
} pmove_t;
```

This looks to be a very straightforward struct (albeit large). The variable that stands out like a sore thumb, however, is the first one, `ps`, which is of type `playerState_t`. You know already that you are attempting to modify a `playerState_t` member of the player (velocity), so by seeing this variable here you have an indication of being on the right track. The second member declared is `cmd`, of type `usercmd_t`. Aha! You already made a note to research the `usercmd_t` struct, so it is very interesting that you find it here in `pmove_t` as well. The plot thickens. . . .

The struct `usercmd_t` is declared near the bottom of `q_shared.h`, on line 1213:

```
// usercmd_t is sent to the server each client frame
typedef struct usercmd_s {
    int         serverTime;
    int         angles[3];
    int         buttons;
    byte        weapon;           // weapon
    signed char forwardmove, rightmove, upmove;
} usercmd_t;
```

Hmmm, the C comment suggests that this struct is sent to the server every client frame, which you're already looking at by peering through code in `ClientThink_real`. This is a very important clue in the mystery of your missing `client` variable. Now that you know what is contained within these structs, let's jump back to `ClientThink_real`.

Starting around line 841, some C comments begin to suggest that something is being prepared for—a *pmove* or “player movement.” After scanning a few more lines of code, you see the following key snippet:

```
pm.ps = &client->ps;
pm.cmd = *ucmd;
```

In the first line, the `pm` variable's `ps` member is being set to the memory address of the client's `playerState_t` struct. By doing an assignment to a memory address, the `pm.ps` variable essentially *becomes* the client's `playerState_t` variable, and any reads or writes that occur to `pm.ps` will ultimately happen to the client's `playerState_t`. This is very important, so let me reiterate: By setting `pm.ps` to the memory address of `client->ps`, you create a reference to `client->ps`, and any changes made to `pm.ps` will also happen to `client->ps`.

The second line sets `pm.cmd` equal to the pointer to a `usercmd_t` type that was created at the top of the `ClientThink_real` function. And just what is `ucmd` pointing to, anyway? Line 750 answers that question:

```
ucmd = &ent->client->pers.cmd;
```

Interesting . . . this line of code says that the pointer declared earlier is set to the memory address of `ent->client->pers.cmd`. By assigning to a memory address you know you're creating another reference directly to the `client` variable, and in this case, you point to the player's commands, which have been sent to the server. This could be any command, like joining a team, firing a weapon, or—you guessed it—jumping.

## The Move to Pmove

After a bit more logic to check some last-minute things, the final bombshell drops on line 922:

```
Pmove (&pm);
```

Here, the entire `pm` variable construct is passed to a function called `Pmove`. Notice that it is passed by reference, meaning that the actual memory address for the `pm` variable is passed to `Pmove`. Whatever magic happens in `Pmove` is ultimately going to happen to `pm` in this function, and you already know what that means: The change will cascade back to the player's `client` var. Let's visit the `Pmove` function now and find out what's happening.

Searching through the code base, you find `Pmove` situated nicely at the bottom of `bg_pmove.c`:

```
void Pmove (pmove_t *pmove)
```

You're hot on the trail. `bg_pmove.c` is the file that also holds `PM_CheckJump`, the original function you found that held the modification to the player's z-axis velocity. By looking at this function, you can see that `Pmove` takes a pointer to a `pmove_t` variable to be passed in (in this case, it is referred to within the function body as `*pmove`). The `Pmove` function then calls a function called `PmoveSingle`, handing off the same `*pmove` variable, on line 1816:

```
PmoveSingle( pmove );
if ( pmove->ps->pm_flags & PMF_JUMP_HELD ) {
```

```
        pmove->cmd.upmove = 20;
    }
```

Interestingly enough, `PmoveSingle` is called first. Then, a check is made to see if the `PMF_JUMP_HELD` flag is on, effectively queuing the actual jump event until the next frame process. The signal for the jump event in this case is the setting of the `pmove->cmd.upmove` member to 20, which as you can guess, is actually setting the player's `client->pers.cmd.upmove` value to 20.

You're very close now. On the very first line of `PmoveSingle`, a global variable called `pm`, which is declared way up at the top of this file, is set to the pointer passed in:

```
pm = pmove;
```

Because `pm` is a global variable, it is now available to every other function in this file. `PmoveSingle` goes on to perform some logic, and then calls a function called `PM_WalkMove` on line 1970. Leaping up to line 672 reveals the contents of the `PM_WalkMove` function; lo and behold, look at what is found on line 690:

```
if ( PM_CheckJump ( ) ) {
```

There, `PM_CheckJump` is called, in all its glory. Heading back up to `PM_CheckJump` on line 339, you can see several lines down that if `pm->cmd.upmove` is less than 10, a jump is considered to have not taken place, and the function exits. Because you queued it up with the last frame pass to 20, however, it will register a jump.

Give yourself a pat on the back—you have successfully solved your own problem. You have tracked down the passing of the client's `playerState_t` variable all the way back to `PM_CheckJump`. Because `pm` is a global variable declared in this file, any function, including `PM_CheckJump`, has access to it. And because `pm` points to the `pmove` pointer passed into the function `Pmove` (deep breath), which in turn holds a `playerState_t` member that points to the client's `playerState_t`, you can be assured that anything affecting `pm.ps` will cascade all the way back to `client->ps`.

As for why this particular functionality is implemented through `Pmove` . . . well, I can't give all my secrets away just yet, can I? I'll discuss just what exactly a `Pmove` is, and why it is important in this instance, in Chapter 5.

**TIP**

In Chapter 3, I showed you how to search for a keyword across your code, when you used VC++'s Find in Files feature to look for `CG_ShotgunPattern`. There is, however, another way to perform a lookup on a specific term. Simply right-click on the text and select **Go To Definition Of** (your selected text). This makes VC++ do the dirty work by searching across all the projects for you for where the variable was originally defined. If the variable definition is found, VC++ opens the file and jumps to exactly the line where the variable definition is located. If you have more than one variable with the same name, you may see a dialog box allowing you to select which specific variable you want to find. Sometimes this method requires you to build a browse-directory first, so if you get a warning that tells you there is no browse info available, simply allow it to build the info, and the search should complete.

## Modifying Jump Velocity

The benefit of having waded through the code is that you now have a deeper understanding of how variable constructs are passed from function to function within the game. This will be key when you start to modify `cgame` code in greater detail in Chapter 6. For now, let's try tweaking the value of that jump velocity based on the player's health.

Let's make a change to the `PM_CheckJump` function that uses the following rules:

- If the player has more than 100 health, let him jump 25% higher.
- If the player has more than 30 health, give him a normal jump.
- If the player has less than or equal to 30 health, cut his jump height in half.

That's pretty easy; I think you can implement it with the following snippet of code. Replace the line that actually sets the `velocity[2]` to `JUMP_VELOCITY` with the following lines:

```
if (pm->ps->stats[STAT_HEALTH] > 100)
```

```
pm->ps->velocity[2] = JUMP_VELOCITY * 1.25;
else if (pm->ps->stats[STAT_HEALTH] > 30)
    pm->ps->velocity[2] = JUMP_VELOCITY;
else
    pm->ps->velocity[2] = JUMP_VELOCITY * 0.75;
```

Give that a compile, and try it out. You should notice a significantly higher jump right from the get-go. Then, fire a rocket into the ground a few times to decrease your health and try jumping again. As your health lowers, so too does your jump height. In fact, when you get below 30 health, you can barely even leave the ground!

## Giving the Player a Jetpack

Now that you know how to modify the player's speed, jump height, and the effect that gravity has on him, how can you tie all of these effects together into a mod? The answer is to give the player a jetpack. Some of *Q3*'s levels, especially those found in the Team Arena Expansion Pack, can be quite large, which makes the addition of a jetpack quite appropriate. To develop this jetpack mod, let's outline what you want it to accomplish:

- The jetpack will function as a booster, which propels the player into the air via the use of an on/off toggle.
- Real-world physics will apply to the player when he is airborne. That is, if he is moving, he will continue to move in the same direction until he hits another object.
- The player will be unable to control his movement until he lands again.

These rules make for a fairly quick and uncomplicated implementation. So, without further ado, let's get started.

## Creating a New pmove Flag

Your first change will take place in `bg_public.h`. In order to assign a new movement type to the player (specifically, flight via a jetpack), you need to create a new pmove flag to accommodate the check. Conveniently, there is a missing flag from the list on line 122 of `bg_public.h`. Go ahead and change the list so that it includes your new flag, as shown here:



```
// pmove->pm_flags
#define PMF_DUCKED 1
#define PMF_JUMP_HELD 2
#define PMF_JETPACK 4 // jetpacking!!
#define PMF_BACKWARDS_JUMP 8 // go into backwards land
```

This change adds a bit flag called `PMF_JETPACK`, and sets its value to 4, allowing you a new variable to look for, to see if the jetpack is being activated or not.

Next, you'll need to create a new console command to hold your toggle for the jetpack. This will be the command to which you bind a key in order for the jetpack's thrust to be activated (or deactivated). The command is created in much the same way you created it when you bound a toggle to the homing missile in Chapter 2.

Open `g_cmds.c` and scroll down to line 1677 and make the following changes to the `ClientCommand` function:

```
else if (Q_stricmp (cmd, "stats") == 0)
    Cmd_Stats_f( ent );
else if (Q_stricmp (cmd, "jetpack") == 0) // toggle jetpack
    Cmd_ToggleJetpack_f( ent );
else
    trap_SendServerCommand( clientNum, va("print \"unknown cmd %sn\"",
cmd ) );
```

After the entry for the stats command, a new command is set up that looks for the string `jetpack`. If this string is found, a new function is called: `Cmd_ToggleJetpack_f`. This is the function that turns your `PMF_JETPACK` flag on or off. Let's go ahead and do that next.

Jump up to where the `Cmd_Stats_f` function begins, and add this new function before it:

```
/*
=====
Cmd_ToggleJetpack_f
=====
*/
void Cmd_ToggleJetpack_f( gentity_t *ent )
{
    ent->client->ps.pm_flags ^= PMF_JETPACK;
}
```

## Bit Flag Uniqueness

The value of a bit flag must use a unique number that cannot be reached by adding any other combination of flags together, which is why your `PMF_JETPACK` flag is 4. Even though it seems that 3 is also available, consider that a player can theoretically be crouching (`PMF_DUCKED`) and holding the jump button down at the same time (`PMF_JUMP_HELD`). Those two flags added together also equal 3. If that had been the case, and you checked for `PMF_JETPACK` with a bitwise operation, you would have received a value of `true` if the user was activating the jetpack or if the player was ducking and holding the jump button at the same time. Not a good thing!

A nice, simple function for a change! This function does only one thing: toggles your `PMF_JETPACK` flag. So, if it is on, this function will turn the flag off; if it is already off, this function flips the switch and turns it on. Now that you have a flag that's going to be your means of determining whether the jetpack is in use and a command to fire the jetpack, let's create the jetpack's effect on the player.

## Defying Gravity

Begin by opening `bg_pmove.c`, the friendly file that contains the movement code for the player. You may recall that `PmoveSingle` was the function used to hand off the player's movement state to various movement functions, such as `PM_CheckJump`. This is a perfect spot for you to check the `PMF_JETPACK` flag. Scroll to around line 1954, and make the following change:

```
    } else if (pm->ps->pm_flags & PMF_GRAPPLE_PULL) {
        PM_GrappleMove();
        // We can wiggle a bit
        PM_AirMove();
    } else if ( pm->ps->pm_flags & PMF_JETPACK ) {
        // jetpacking around
        PM_JetpackMove();
```

```

} else if (pm->ps->pm_flags & PMF_TIME_WATERJUMP) {
    PM_WaterJumpMove();
}

```

Here, after the line of code calling `PM_AirMove`, you perform a check for the `PMF_JETPACK` flag using a simple bitwise operation. Note here that you're using `pm.ps`, the global variable that points to the `client->ps` struct, so—in case you had any doubts—you are truly looking at the player. If the `PMF_JETPACK` flag is found to be on, call a new function, `PM_JetpackMove`, which will be a new movement function that you'll write next.

## A Surprising Effect

Before you add `PM_JetpackMove`, let's review what you need to do. The effect of a jetpack being turned on should modify the player's z-axis velocity; that is, it should propel the player upward. Because this will be an effect that takes place over time, you will want to continually increment the player's velocity. However, it's probably not a good idea to blast the player into space, so a fixed maximum velocity should never be breached. This all seems simple enough, so let's write the function in, right above `PM_CheckJump`:

```

/*
=====
PM_JetpackMove
=====
*/
static void PM_JetpackMove(void )
{
    pm->ps->velocity[2] += 10;
    if (pm->ps->velocity[2] > 250)
        pm->ps->velocity[2] = 250;
}

```

This looks like it should work without any adjustments. You simply increment the player's z-axis velocity by 10 each time this function is called (which will happen repeatedly, as long as the `PMF_JETPACK` flag is on). And, just for safe measure, if the player's z-axis velocity ever goes beyond 250, it will be capped back to 250. Let's compile it all, and give it a whirl.

After you drop your new `qagamex86.dll` into your MyMod folder and fire up a *Q3* map, set up your jetpack key binding by typing the following in the console and pressing Enter:

```
\bind j "jetpack"
```

You can substitute `j` with whatever you wish; I used my right-mouse button, which is `mouse2`. Pressing the button bound to the jetpack command should turn the jetpack on, giving you lift; another press of the button should turn it off. Something is amiss, however. A few moments after leaving the ground, your player vibrates violently in place, and never gains more than a few inches of height. What's going on? You have followed all the rules, handling the velocity as an increment across time, yet for some reason your player won't stop this hovering dance. This brings up the question: If the jetpack doesn't work, how the heck does the flying powerup—among the most sought-after powerups in *Q3*—work?

## Borrowing Code from PM\_FlyMove

In *Q3*, there is a powerup that can be grabbed by the player that allows him to fly. The player is given complete freedom from gravity (for a limited time), and can move in any direction he chooses. Figure 4.1 shows the much sought-after flight powerup in `q3dm19`, *Apocalypse Void*.

If similar functionality already exists in *Q3* to allow the player to ignore gravity for a small amount of time, surely you can implement a jetpack. Let's take a look at what function is called when the player is holding the flight powerup.

Jumping back down to `PmoveSingle`, near line 1968 you see the following bit of code:

```
if ( pm->ps->powerups[PW_FLIGHT] ) {
    // flight powerup doesn't allow jump and has different friction
    PM_FlyMove();
}
```

It looks as though the `PW_FLIGHT` flag in the powerups array of the player's state is the flag that maps to the flight powerup in *Q3*. Here, the code says, "If the `PW_FLIGHT` flag is in existence in the player's



**Figure 4.1** *The flight powerup*

powerups array, call a function called `PM_FlyMove`.” Scroll up to the definition of `PM_FlyMove`, and see what magic is happening there. You should see `PM_FlyMove` around line 553; it follows below.

```

1. static void PM_FlyMove( void ) {
2.     int         i;
3.     vec3_t      wishvel;
4.     float       wishspeed;
5.     vec3_t      wishdir;
6.     float       scale;
7.
8.     // normal slowdown
9.     PM_Friction ();
10.
11.     scale = PM_CmdScale( &pm->cmd );
12.     //
13.     // user intentions
14.     //
15.     if ( !scale ) {
16.         wishvel[0] = 0;

```

```
17.         wishvel[1] = 0;
18.         wishvel[2] = 0;
19.     } else {
20.         for (i=0 ; i<3 ; i++) {
21.             wishvel[i] = scale * pm1.forward[i]*pm->cmd.forward-
move + scale * pm1.right[i]*pm->cmd.rightmove;
22.         }
23.
24.         wishvel[2] += scale * pm->cmd.upmove;
25.     }
26.
27.     VectorCopy (wishvel, wishdir);
28.     wishspeed = VectorNormalize(wishdir);
29.
30.     PM_Accelerate (wishdir, wishspeed, pm_flyaccelerate);
31.
32.     PM_StepSlideMove( qfalse );
33. }
```

Let's dissect this bit by bit. An initial function, `PM_Friction`, is called to apply appropriate physics to the player (line 9), which causes a gradual deceleration if the player isn't applying movement force in some direction (walking forward or backward, jumping up, and so on). You didn't use this function, but it might be a good idea to try it to get some more realistic response for the player while airborne.

Next, `PM_CmdScale` is called, passing in the player's command variable (line 11), which would point to any keys or buttons that the player may be pressing at the time. The function returns a float, which is assigned to a variable called `scale`, which, for the record, can be any value between  $-127$  and  $127$ . At first it may not seem make sense to return the movement key press as a float (either the player is moving forward or not, right?). The reason it is done this way is so that a player using analog controls can adjust the amount of speed he wishes to apply to a given direction.

If no movement is detected by the player's control, the `scale` variable is returned as a 0, which tells the `PM_FlyMove` function to set a temporary velocity vector, `wishvel`, equal to 0 on all three of its axes. If movement is detected, `wishvel` obtains the player's forward- and right-movement degrees and multiplies them by the scale detected (line 21). Then, the player's upward movement is also taken into

**NOTE**

There are two types of controls in most games played today. *Digital* controls are those that are registered by a game as either on or off, like a toggle. A good example of a digital control is a key press on the keyboard. The second type of control is *analog*, which indicates that there is a range or degree of selection being transmitted to the game. The best example of an analog control is a steering wheel, used for racing simulations. By turning the wheel gradually, a small degree of a turn is sent to the game to appropriately adjust the virtual car. As the steering wheel is turned more sharply one way or the other, a higher-degree turn is transmitted, thereby resulting in a sharper turn in the game.

consideration, and added (by scale) to the z axis of `wishvel` (line 24). A call to `VectorCopy` is made to copy `wishvel` into another vector, `wishdir` (line 27), which is then sent to `VectorNormalize` to obtain a measurement of speed (line 28).

Next, `wishvel` and `wishdir` are passed to `PM_Accelerate` (line 30), a function used to apply acceleration to a player object, with `pm_flyaccelerate` passed as the intended acceleration factor (`pm_flyaccelerate` is declared at the top of `bg_pmove.c`). Then, a final function is called: `PM_StepSlideMove` (line 32). This call poses the most interesting content of all, as it takes one input parameter, aptly named `gravity`. In the call from `PM_FlyMove`, this value is set to `false`. Hmm . . . `gravity` set to `false`. This sounds very much like a solution you want to implement.

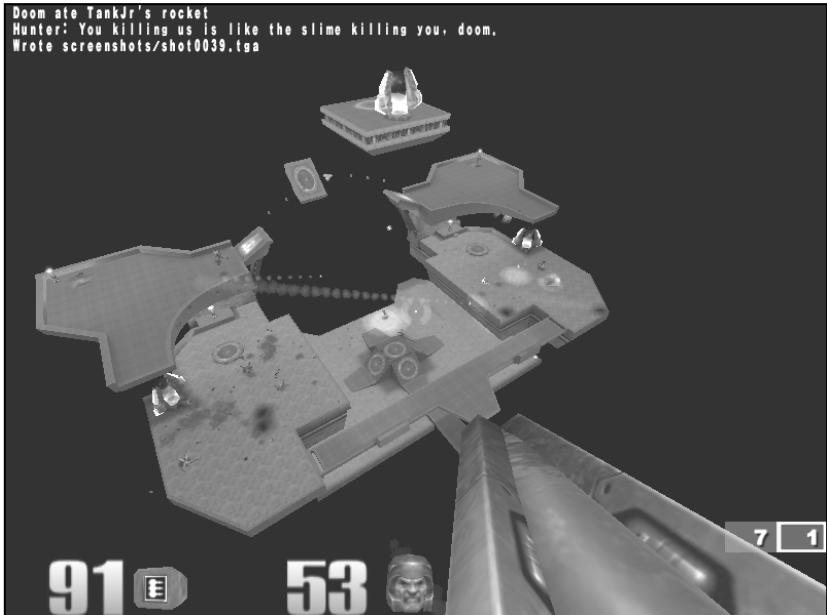
Let's go back to `PM_JetpackMove` and add this same function call to the end. The new function looks like this:

```
static void PM_JetpackMove( void )
{
    pm->ps->velocity[2] += 10;
    if (pm->ps->velocity[2] > 250)
        pm->ps->velocity[2] = 250;
```

```
PM_StepSlideMove( qfalse );  
}
```

Compiling and running this a second time reveals some more predictable results; when activated, the player begins to accelerate into the air, free from gravity. If the key bound to the jetpack command is pressed a second time, the jetpack is disengaged, and the player falls freely to the surface (getting hurt if he falls too far). Figure 4.2 shows a bit of fun the player is having by jetpacking above `q3dm17` and firing away at some unsuspecting bots.

For now, you can say that you're satisfied with the implementation of the jetpack; it meets the criteria listed at the beginning of this section. After the player leaves the ground, he continues moving in the direction he was moving during liftoff until he hits a surface (or is killed). While in the air, he also loses the ability to change directions (which makes sense, because his feet aren't on the ground), making for a more realistic jetpack. You may want to tweak the jetpack's functionality by borrowing more code from `PM_FlyMove`, but for now, let's move to another aspect of dealing with the player.



**Figure 4.2** *The jetpack in action*



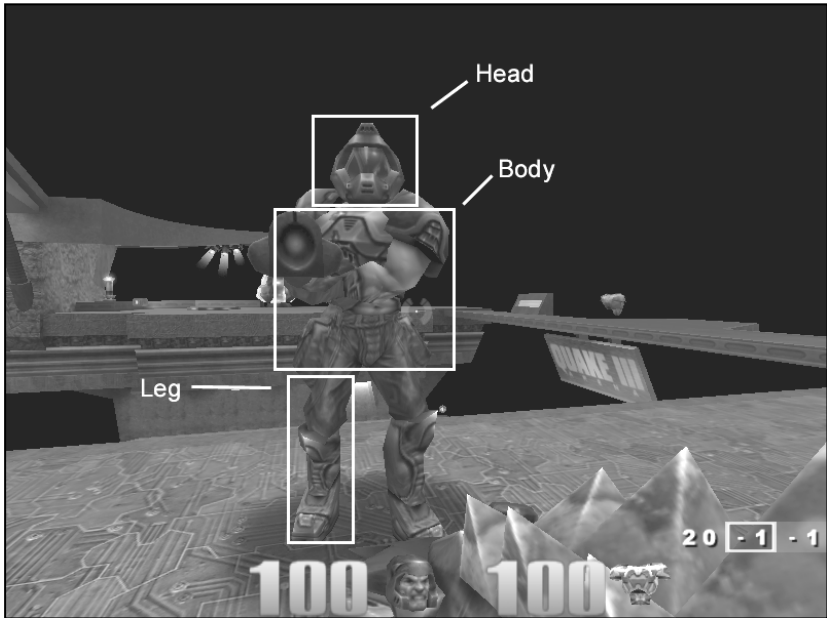
## Implementing Locational Damage

As you get into mod development on a more regular basis, you're undoubtedly going to come across many fans shouting "Realism!" In fact, realism is quite a contention in the world of user-developed mods nowadays, especially when it comes to how much realism in a mod is appropriate. Granted, having futuristic aliens and robots battling each other with plasma rifles isn't necessarily defining the norm of realistic warfare, but on the other hand, certain fundamental issues should not be overlooked. One such issue is damage inflicted to different areas of the body. If you are creating a mod in which strategy and tactics play more of a role than mind-numbing deathmatch, then you may want to consider alternative ways of handling damage done by weapons. Case in point: Mods like *Urban Terror* and *Counter-Strike* use a locational damage system, where players take different levels of damage based upon where they are hit. Leg and arm shots obviously do less damage than a shot to the chest. And, it goes without saying that a headshot almost definitely means instant death. *Q3*, on the other hand, does not use a locational damage system; let's take a look at how one could be implemented.

### Creating Hit Flags

For starters, you need to find a way to determine what part of the player is being fired upon, because there is currently no way in *Q3* to scan the area that is hit. There are definitely a number of ways to handle this, some of which extend beyond the scope of this book and are based on different 3D model types. For now, let's take a look at what makes up the current *Q3* player, and see if it can be broken down in segmented parts. The *Q3* player, as shown in Figure 4.3, is constructed of three identifiable areas: the head, the body, and the legs. This could be further broken down, but for this tutorial, three areas are sufficient; you are certainly welcome to expand upon this as you see fit.

You can assign some bit flags to these body parts, which you can then apply to the actual target once in the game. To do so, open `bg_public.h` and scroll down to about line 244, right after `EF_TEAMVOTED` is declared. Here, set up some new bit flags, with the `LC` naming convention, which seems appropriate for locational flags. Right after



**Figure 4.3** *The Quake III player*

EF\_TEAMVOTED is declared, add the following lines (notice that room is also left for no body part, as in the case of LC\_NONE):

```
// LOCATIONAL DMG FLAGS

#define LC_NONE          0x00000000
#define LC_HEAD         0x00000001
#define LC_BODY         0x00000002
#define LC_LEG          0x00000004
```

Next, you need a new client variable to tell where the player was hurt—in essence, a place to store any or all the flags you have just declared. By now, you should be pretty familiar with what variables the player maintains. Because this variable is something that will expire as soon as the player is killed, and not something that will be updated as frequently as player speed or velocity, the prime candidate for placement is within the `gclient_s` struct. Open `g_local.h` and scroll down to around 276, where you should see declarations of `lastkilled_client`, `lasthurt_client`, and `lasthurt_mod`. Add your new variable after `lasthurt_mod`, as shown in the following code:

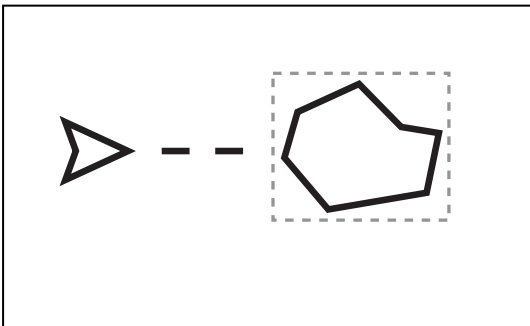
```
int    lastkilled_client; // last client that this client killed
int    lasthurt_client;  // last client that damaged this client
int    lasthurt_mod;     // type of damage the client did
int    lasthurt_loc;     // location of damage the client did
```

The new variable called `lasthurt_loc` holds the bit flags you created as damage is done to specific parts of the target's body. Next, you need to calculate how the target is actually hit. The simplest way that this can be done is by estimating where the legs, body, and head are, based on the player's height. You do this by using the player's `currentOrigin`, `mins`, and `maxs`, which reside in the `entityShared_t` struct.

## The Bounding Box

If you've got a good memory (and if you want me to give you another 50 bonus points), you'll recall that I talked briefly about bounding boxes in Chapter 3. The bounding box is very familiar to game developers because it is the primary tool used in collision detection, which refers to the act in which a game tests whether two objects are touching each other. Think back for a moment to a classic game like *Asteroids*. Remember when those meteors smacked into your ship, causing it to explode? That was a very simple form of collision detection using a bounding box. Likewise, each asteroid had its own bounding box; when your tiny ship blasted away at an asteroid, its bounding box assisted in the collision detection, as shown in Figure 4.4.

The dashed square surrounding the asteroid represents the invisible bounding box in the game. You never saw that box while playing, but the code controlling the game knew that it was present for each and every asteroid. As a player fired a shot toward the asteroid, the code



**Figure 4.4** The bounding box in *Asteroids*.

for the game would test to see if a bullet passed into the box. If it did, the computer would register a hit and the asteroid would break into smaller asteroids. As scary as it may sound, game programmers are still using those same bounding-box methodologies today. Hey, if it ain't broke, why fix it?

In *Q3*, there really is no difference in the usage of the bounding box. Certainly, things are a lot more complicated now that you are in a virtual 3D world, but fundamentally the rules are the same. In *Q3*, mins and maxs vectors define an entity's bounding box. There is only a need for two vectors, because each vector spans three directions in 3D space, and if both vectors point to

each other along one plane, all the coordinates hook to each other to form a complete rectangular cube in 3D space.

Now that you know your player has a bounding box associated with it, how do you find out its height in order to properly gauge a headshot as opposed to a shot in the leg? At the top of the `g_client.c` file, two static vectors are declared as global variables:

```
static vec3_t   playerMins = {-15, -15, -24};
static vec3_t   playerMaxs = {15, 15, 32};
```

If you exclude the z axis (because you are not concerned with how thick or thin the player is), the distance between the two farthest points (-15, -15) and (15, 15) is 30, 30 (which equates to 30 units wide by 30 units tall). You can see that these units are then poured into the player's variables when the player is first spawned in the game at line 1147 in `g_client.c`, in the function `ClientSpawn`:

```
VectorCopy (playerMins, ent->r.mins);
VectorCopy (playerMaxs, ent->r.maxs);
```

Here, the units representing the player's bounding box are passed into the `ent->r.mins` and `ent->r.maxs` values, respectively, which can then be used from other parts of the game. Because you have a height of 30 to play with, you can estimate that the head area is roughly 8 units from

## NOTE

There are actually two more coordinates that *Q3* applies to all entities in the game, `absmin`, and `absmax`. They are used when the entity in question is being rotated to get a more accurate bounding box—at the cost of a bit more processing.

the top of the player, while the body extends another 18 units down from there. The remaining 4 units will be considered a leg shot.

## Scanning Body Parts

Now let's write the function that will actually perform the locational damage check. Start by declaring the function. Back in `g_local.h`, scroll to around line 480, where `G_Damage` is declared; underneath it, add the following line:

```
int G_LocDamage (gentity_t *targ, gentity_t *attacker, int damage,
vec3_t point);
```

Your function is called `G_LocDamage`, and will require the following entities passed to it:

- The target (the player who is being fired upon)
- The attacker (the player who is actually doing the shooting)
- The damage
- An entry point in the 3D world

The new function will also return an integer equal to the amount of damage that was caused. You can add the function just below `G_Damage` in the `g_combat.c` file. Open `g_combat.c`, scroll to where `G_Damage` ends (just about line 1049), and add this new function:

```
1. /*
2. =====
3. G_LocDamage
4. =====
5. */
6. int G_LocDamage(gentity_t* targ, gentity_t* attacker, int damage,
vec3_t point) {
7.
8.     int targHeight;
9.     int feetAt;
10.    int inflictorHeight;
11.
12.    if (!damage)
13.        return 0;
14.
15.    feetAt = targ->r.currentOrigin[2] + targ->r.mins[2];
16.    targHeight = targ->r.maxs[2] - targ->r.mins[2];
```

```

17.     inflictorHeight = point[2] - feetAt;
18.
19.     if (inflictorHeight > targHeight - 8)
20.         targ->client->lasthurt_loc |= LC_HEAD;
21.     else if (inflictorHeight > targHeight - 26)
22.         targ->client->lasthurt_loc |= LC_BODY;
23.     else
24.         targ->client->lasthurt_loc |= LC_LEG;
25.
26.     switch ( targ->client->lasthurt_loc )
27.     {
28.     case LC_HEAD:
29.         damage *= 3;
30.         trap_SendServerCommand( attacker-g_entities, "print "Head
Shot!\n" );
31.         break;
32.
33.     case LC_BODY:
34.         damage *= 1.2;
35.         trap_SendServerCommand( attacker-g_entities, "print "Body
Shot!\n" );
36.         break;
37.
38.     case LC_LEG:
39.         damage *= 0.6;
40.         trap_SendServerCommand( attacker-g_entities, "print "Leg
Shot!\n" );
41.         break;
42.
43.     }
44.     return damage;
45.
46. }
```

Let's break this function down in order to understand what's going on. Initially, the value of the passed-in damage variable is checked to see if it is actually a positive value. If the damage is less than or equal to zero, control of the code is handed back to the calling program that runs the `G_LocDamage` function (line 12). If there is damage, however, the function continues.

## NOTE

`G_LocDamage` is based on another height-level locational damage system, created by Arthur "Calrathan" Tomlin.

The next thing `G_LocDamage` does is try to figure out what the target's height is, because it only has absolute x, y, and z coordinates for the target. By *absolute coordinates*, I mean that when `Q3` looks at a value for x, y, and z on a player's location, it retrieves numbers that map to an actual location in the world, not the height, width, and depth of a player. What you need to do is take these absolute values and perform simple subtraction to get the *relative* x, y, and z coordinates—in other words, how the x, y, and z coordinates apply to a certain character.

## My Feet Are . . . Where Now?

Start by creating a variable called `feetAt`, and assign it the value of your target's origin, or `r.currentOrigin` (line 15), along the z axis (because you will be traveling the axis up and down to determine height). Then, add the target's `mins` value to the current origin, which gives you the z-axis location of the target's feet. The `mins` value is added, instead of subtracted, from the center point because the `mins` value is a negative number.

Your second variable, `targHeight`, is set to the value of the target's `maxs` value minus the target's `mins` value (both along the z axis). This gives you a total height of the player (line 16). The third variable, `inflictorHeight`, is then assigned the value of the bullet minus the vertical location of the player's feet, which is stored in `feetAt`. This gives you a relative height of the bullet as it hits the player, which you can compare to the actual player's height to estimate what part of the body was hit (line 17).

Because you already know that the player's bounding box is 30 units tall, you can start to play with some numbers to create the effect of locational damage. Approximately, the first 8 units from the top of the player down are a good range for a headshot. The code reflects this, as it checks to see if the height of the bullet (`inflictorHeight`) is greater than the target's height, minus 8 units (line 19). If so, you have a valid headshot, and you can apply the `LC_HEAD` flag to your `lasthurt_loc` variable. If the bullet didn't hit the player 8 units from the top (or less), the next check is to see whether the bullet came anywhere within the top 26 units of the player (recall that you estimated the body's height being another 18 units). If so, this will be considered a body shot, and the appropriate `LC_BODY` flag will be applied (line 21). Finally, any remaining registered hit below the 26-unit mark will be flagged as a leg shot.

## Switching Off the Hits

Once you have a hit flag in place for an appropriate body part, you can switch off the value. A *switch* is a standard C-language construct that allows you to perform a number of operations based on a single value, if that value can have multiple results. Think of it as a giant *if-then-else* block. Because your hit flag can be one of three different values, a switch is an efficient way of handling the various bits of logic, and allows for easy addition of new cases in the future.

For the case of a headshot (line 28), multiply the damage by 3 and send a text message back to the client telling him that a headshot has occurred. For a body shot (line 33), multiply the damage by 1.2 and send another message, letting the player know a body shot was made. Finally, for a leg shot (line 38), multiply the damage by 0.6 and send a message acknowledging the leg shot. Because `G_LocDamage` is a function that returns an integer, make sure it is being passed back; wrap the function up by returning the newly updated damage value.

The last part you need to implement is the actual call to `G_LocDamage` in order to apply your new calculations to the attack being made. This is done within `G_Damage`, which you should remember from Chapter 3. Make sure you are still in `g_combat.c`, and scroll up to about line 1016, where you should see some code checking for `if (targ->client)`. Go ahead and make the following changes, so that the function call looks like this:

```
if (targ->client) {
    // set the last client who damaged the target
    targ->client->lasthurt_client = attacker->s.number;
    targ->client->lasthurt_mod = mod;

    // if target is applicable, apply locational damage
    if (targ && targ->health > 0 && attacker && take && point)
        take = G_LocDamage(targ, attacker, take, point);
    else
        targ->client->lasthurt_loc = LC_NONE;
}
```

Nothing really complicated is happening here; all that occurs is a check to make sure that the target is valid, it isn't dead, the attacker is valid, there is a valid amount of damage inflicted, and there is a valid point of entry for your inflictor (the bullet). If all of these checks succeed, the variable that holds the current total damage, `take`, is assigned



**Figure 4.5**

*A successful headshot on a Q3 bot*

to the result of the function call. If the checks fail, locational damage is not applied. Instead, the `lasthurt_loc` variable is assigned the flag `LC_NONE`. That should be it! Give your code a compile and run-through. Test it on a few bots and see if you can make some headshots. It can be quite challenging, especially when the bots move quickly. The locational-damage code in action is pictured in Figure 4.5.

## Summary

Hopefully, you've come away from this chapter with some knowledge about the player and how it interacts with the *Q3* world. By understanding how the player moves, you will be able to more efficiently modify things like player speed, velocity, acceleration, and so on. This will help if you ever decide to break the player up into multiple classes within your mod, each class having different movement attributes. Also, by working with locational-damage code, you have a better understanding of how collision detection works in the world of *Q3*, which is helpful when you are trying to visualize how the player's dimensions are represented.

I hope some of the code-research techniques I introduced to you in this chapter will assist you in your further exploration of the code base. In the next section of the book, you'll push those methods even further as you explore the interaction between server-side code, client-side code, and user-interface code.



# CHAPTER 5

# QUAKE COMMUNICATION

In the first section of this book, you became acquainted with the *Q3* code base. You also got to make a few modifications to things like weapons and players. Now, you're going to take a bit of a break from coding so that you can get up to speed on what actually goes on behind the scenes when a mod is created. In this chapter, I'll give a broader view of *Quake III*—not just as a game, but as a true Internet application—and the reasons it is constructed in a modular fashion.

## The Client/Server Relationship

Although *Q3* can be played in single-player mode, in which one human takes on a contingency of bots, this is not the ultimate functionality of the game. Where *Q3* shines is in its capability to connect to itself over a network, typically a large area network (LAN). A LAN is simply a group of computers that can “talk” to each other. When a computer is part of a LAN, it can share resources, such as files or a single printer, with other computers on the LAN.

Computers can also be a part of a wide area network (WAN), which consists of computers that span a large geographical area, but still communicate with each other. The best example of this network is the Internet, in which hundreds of thousands of computers are able to “jack in” and gain access to resources such as Web sites, ftp servers, news feeds, and so on. If you have a connection to the Internet on your computer, you are most definitely participating in a WAN. *Q3* can also utilize a WAN for online playing capabilities, allowing players to fight against one another even if they are separated by thousands of geographic miles.

When computers are part of a network they are granted the ability to run various types of applications on that network. For example, if you spend any amount of time on the Internet you probably use an email program to send and receive electronic mail and a Web browser to

surf the Web. These are examples of *client applications*; that is, they reside on your computer, or *client*, and make data requests to another computer, the *server*.

As it happens, *Q3* is also a client application. When your computer is connected to a network and you launch *Q3*, you can search the network for other *Q3* games currently in progress. If you find one, you can jump right in and start fragging away. Alternatively, you can launch a game of *Q3* and set up a game yourself, awaiting other players to join you for a deathmatch. In this scenario, *Q3* acts as a *server application*—that is, a network application awaiting client connections so that it can distribute data accordingly. These abilities make *Q3* a fully functional *client/server application*.

### NOTE

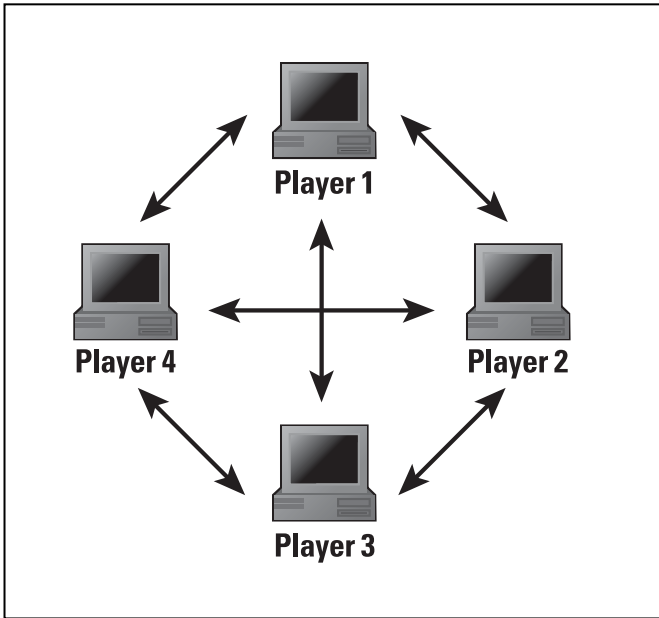
Even when you launch *Q3* in single-player mode, it still acts as a client/server application because you are technically connecting to your own computer.

## Leaving Peering in the Past

The original *Quake* was id's first attempt at implementing true client/server architecture. With its *Doom* series, id had used a quicker but less robust architecture, often referred to as the *peer-to-peer* model. In a peer-to-peer network, a computer sends information to every other computer on the network, as shown in Figure 5.1. Because this model causes quite a bit of network traffic, it is better-suited for small groups of computers; that's why *Doom* allowed only a maximum of four people to play simultaneously on a network.

In addition to boosting network traffic, peer-to-peer networks offer less flexibility when it comes to actually running the game. For example, all computers participating in a game of *Doom* had to be prepared beforehand, with everyone starting at the same moment; there was no way for a computer to jump in and join a game already in progress. Even worse, if one player's computer had a slower modem (say, a 9.6Kbps modem as opposed to a 28.8Kbps modem), all the computers in the game would be slowed to that speed.

As *Quake* was being developed, the need for a client/server model was evident. Using client/server architecture, one computer would act as

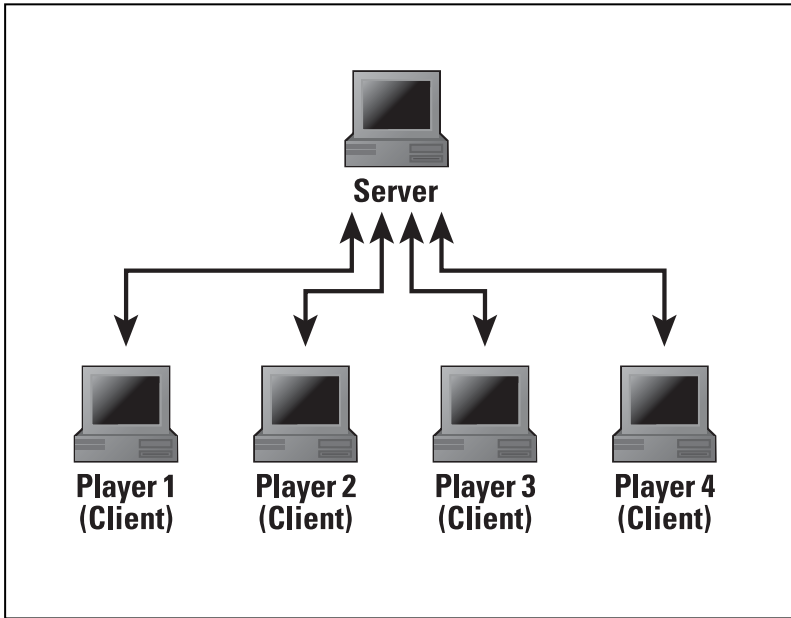


**Figure 5.1** A peer-to-peer network

the server, or *host*, doing nothing but distributing data to the clients that connected to it. Then, as clients joined, they would receive information from the host as to what was happening in the game—who had which score, who was shooting whom, and so on. The only thing the client needed to do was sit back, receive the updates, and play away. This architecture would result in a lot less network traffic, because clients only ever talked to one computer: the server (as shown in Figure 5.2).

As an added bonus, the client/server model allowed for clients to freely join and leave at will, because they were no longer responsible for “explaining their data” to each and every other computer on the network. A client could connect, and leave it to the server to notify the other computers on the network. Likewise, when a client disconnected the server could send an appropriate message to the remaining clients. This, clearly, was a much better model. And so, *Quake* was built, the players rejoiced, and on the seventh day, id rested.

There was one problem, however: latency.



**Figure 5.2** A client/server network

When a *Quake* server needs to send an update to a client, it wraps all the various bits of data that are important into something called a *packet*, which is transmitted over the network. A packet can contain anything, such as a player's location, his currently selected weapon, what level he's on, how much health he has, and so on. As more data is needed, the size of the packet grows. There is also some additional overhead to take into consideration, such as information about when a packet starts and when a packet ends. As a result, packets can become very large—and as packet size grows, so too does the amount of time it takes to deliver it. If the delivery time exceeds an acceptable amount, a delay, or *pause in game time*, occurs. That is, the client expects data from the server so that it can update the game, but the data does not arrive quickly enough. On the client's end, the result is delayed or jerky movement, and the seemingly random appearance and disappearance of missiles. Players can even be killed without ever seeing an enemy! The term used to describe this delay in time experienced from the client to the server is referred to as *latency*. Many players have labeled the effects this delay produces in the game as *lag*.

## Lag in a Nutshell

Imagine there are two people playing a game of *Quake* on a network. Player A is on a high-speed connection to the Internet, so his computer can talk to the server in tens of milliseconds. Player B is on a 28.8Kbps modem, which receives data only at about 3.6Kbps. In a current frame of game time, Player A is standing across the room from Player B. The server updates both players, telling them what's going on, who is standing where, and what is happening; when both updates arrive, the two players see that they are facing each other.

Then, Player A decides he is going to fire a rocket at Player B, so he selects his rocket launcher, aims at Player B, and fires. The *Quake* server begins sending out updates to all the clients, alerting them that Player A is firing a rocket at Player B, so they can update their animations and sound effects accordingly. Player B, at that moment, experiences a “hiccup” in the network connection. It could stem from any kind of problem, like some bad data being received on the modem, or just a poor ping to the server. Regardless, the packet holding the data that says “Player A is firing a rocket at Player B” does not arrive.

---

### Losing Data

---

Thanks to the protocol that Q3 uses to transmit packets in a game, User Datagram Protocol (UDP), packets can indeed be lost or *dropped* en route from the client to the server.

Despite this glitch, UDP was chosen because less overhead is required to transfer UDP packets, and speed is definitely of the essence in a game like Q3. With the alternative protocol, Transmission Control Protocol (TCP), a true established connection between the client and server is required, and packets that are lost are also re-transmitted; coding around these features makes for an unnecessary headache. TCP is more common in applications like FTP (File Transfer Protocol) and HTTP (Hyper-Text Transfer Protocol), which are used in common Internet applications like file-sharing programs and Web browsers, respectively.

Because Player A has a fast connection to the *Quake* server, he sees the results of his attack instantly (the missile moves pretty fast in *Quake!*); his rocket travels the length of the room in a split second, smacks into Player B, and blows him up into tiny gibs. In reality, this involved multiple packet updates: one during the firing of the gun, possibly several during the rocket's traversal of the room, and yet another to indicate that Player B was hit and killed. Meanwhile, Player B experiences every online player's worst nightmare: *lag*. He is frozen while his computer awaits its next update packet from the server to tell his game what is happening.

Finally, the network hiccup resolves, and Player B's connection smoothes out long enough to receive a packet update from the server. Because several seconds have gone by, many packets were lost; as a result, Player B's game simply picks up from the most recent packet it receives. And guess what packet it gets next? "Player B was killed by a rocket fired by Player A two seconds ago." Suddenly, Player B keels over dead in the game, hit by a rocket he never saw or heard coming. There was never even a chance to get out of the way! Player B ends up very frustrated because his *Quake* game had no way of knowing in time what was happening on the server.

Although this is a worst-case scenario, it can happen. The reality of it is that even tiny differences and small delays are enough to throw a client's synchronization off, forcing game play to be jerky, unpredictable, and difficult to maneuver through. Even if Player B never experienced a network hiccup, and continued to receive a steady stream of updates from the server, his modem's limitations wouldn't physically allow the same number of updates per second as Player A's did. Therefore, the game would always tilt in the favor of Player A, who gets a smooth, clean view of the world.

## The Quest for Low Latency

The root of the lag problem was that *Quake* was acting as a *dumb client*, meaning that every single thing that happened in the game would be transmitted through the server. To resolve this problem, the programmers at id devised a way to pass only the information that was absolutely necessary through the server, and have the player's game, on the client-side of things, handle the rest. For example, suppose a player fired a rocket into the air at a given angle. Under the old



model, the client would send 10 updates a second to the server saying, in essence, the following:

1. A rocket has been fired.
2. A rocket has moved to here.
3. Now it is here.
4. Now it has moved to the right a bit.
5. Now it has moved up a little.
6. Now it is going to move a bit farther.
7. Now it has moved even farther still.

Sheesh! What a waste of bandwidth! Wouldn't it make more sense for the client to say the following:

1. A rocket has fired from X position and at Y angle.

If the client had a general understanding of how missile physics works in the game, it could theoretically take a starting position, a firing direction, and be able to calculate everything else on its own. By internally handling all that extraneous data, precious bandwidth is conserved; this can then be used for more crucial communication. The act of the client getting a simpler picture of the game and filling in the rest of the data on its own is called *interpolation*. Because of the success of this new technique, interpolation went on to form the basis of client/server development for *Quake II* and, ultimately, *Q3*.

## The Process of Updates

The interpolation code has been reworked, rewritten, and revised many times since the original *Quake* hit the shelves. *Q3* uses the best of all the ideas that came out of experimenting with *Quake* and *Quake II*. The end result is a very efficient system of transmitting data and making guesses as to what is happening around the player.

When a *Q3* server is active, its responsibility is to deliver the game's state to all the clients participating in the game. It does this by sending updates, often called *snapshots*, to the clients at regular intervals. These updates contain information that tells each client where it is and what it is doing. The updates also include information on the other players on the server, and what sneaky actions they are up to. This is what is known as an *authoritative update*; in other words, the

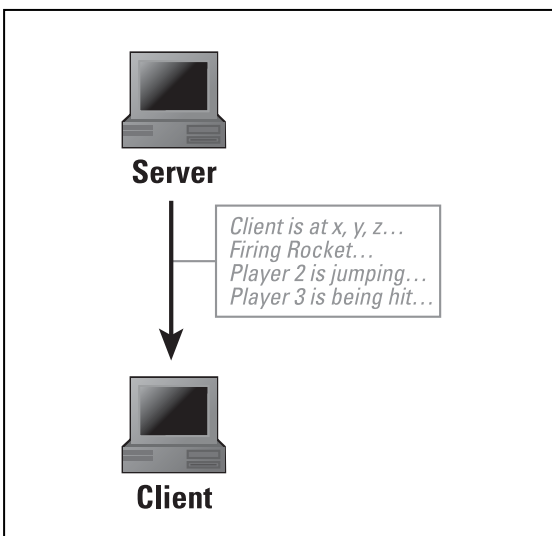
server is making the final decision as to what is happening in the game (see Figure 5.3).

When the client receives an authoritative update, it responds by adjusting everything to match the server's view of the world, such as moving camera angle, positioning other players, and spawning client-side events that should be happening in the game at that time.

Control then moves momentarily to the client so that it can look at localized information specific to its own player, such as whether the player has pressed any controls (forward, back, jump, and so on) that would affect its movement. During this brief interval (usually less than one second), the client-code assesses any commands that the player has issued and uses this information to interpolate where the player will move *to*. This is known as *prediction* (see Figure 5.4). This stage

## NOTE

Technically, the original *Quake* never had the capability to interpolate. Because it was designed as a full dumb client/server model (the client sends *everything* happening back to the server with each update), it never contained any code to make guesses or fill in missing data. When id Software began the task to smarten up the *Quake* client, they broke the project out into a completely separate application, dubbed *QuakeWorld*.



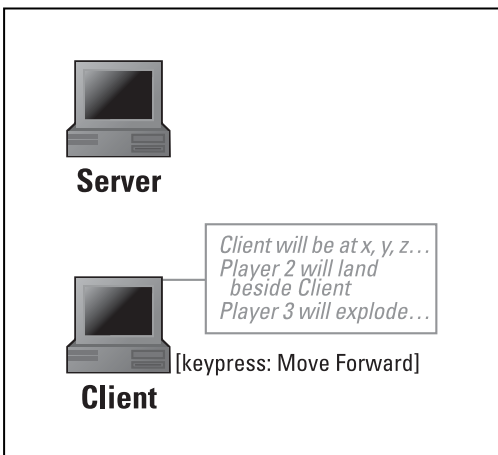
**Figure 5.3** The authoritative update

allows the client to offset some of the data transmission needed to keep the server in constant sync. Instead, it creates a sort of pseudo synchronous state; the client has about a pretty good estimation of what is going on, with a small margin of error.

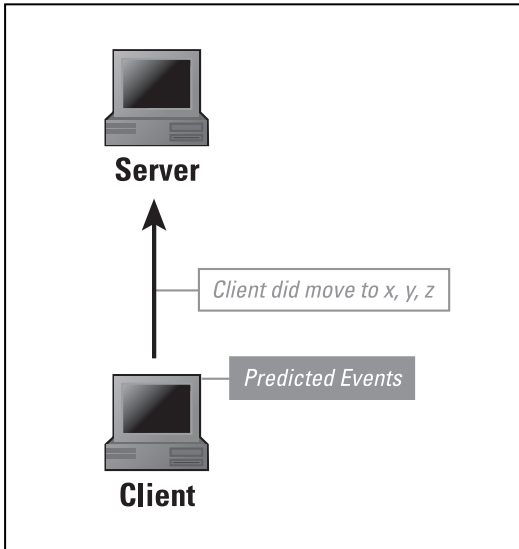
When the client has interpolated all the moves and events that the player will see, it sends the important parts of that data back to the server. The important parts are the non-predicted, real movements and actions that were entered into the game by the player, called *pmoves* (remember all the pmove research you did back in Chapter 4?). The server knows to trust these updates as accurate, because it is handling an individual pmove for each client involved in the game. All the other predicted information is kept separate from the update packet (see Figure 5.5).

When the pmove update completes, the server looks at all the updates from each client, reorganizes them, and distributes a new snapshot of the game to the clients, thus starting the process over again. Because the client already had predicted information from the second stage, there is an expected adjustment as each authoritative update comes in from the server. This is where the phrase “The Server is God” comes from; no matter what the client truly believes is happening in the game at any one time, the server will be sure to make the final call.

Knowledge of the client/server model and how it applies to *Q3* is essential for developing mods. By understanding what data is authori-



**Figure 5.4** *The prediction*



**Figure 5.5** *The pmove update*

tative and must be sent to and from the server, you can make judgment calls as you develop your code. It will become easier as time goes on for you to know what data can safely be predicted, and not affect the outcome of the game (much) if it is overwritten by data sent from the server.

Now that you have an understanding of how the communication between a *Q3* client and server works at the network level, let's take a look at the communication between the three modules of *Q3*'s code, *game*, *cgame*, and *ui*.

## Bridging game, cgame, and ui

The *Q3* code base is broken out into three modules. There is the *game* project, which deals with all the server-side logic. The *cgame* project handles the client-side aspect of things, such as sounds, graphical events, and such. Finally, there is the *ui* project, which allows for the support of the user interface, including menus. You took a brief look at a list of *game* and *cgame* files in Chapter 2; here you'll take some time to see what role each of these modules plays.

**NOTE**

As I stated earlier, even though you technically have four projects in VC++ while working with the *Q3* source, the `q3_ui` and `ui` projects are both user-interface projects. The `q3_ui` project is used for the interface to plain *Quake III Arena*, while the `ui` project is used for the Team Arena Expansion Pack. When I refer to the “ui project” from here on in, I’ll be talking about the files found in the `q3_ui` folder.

## The Server Is God

The *Q3* server-side module of code, or game code, has the largest responsibility of all three parts. Because you have been working almost exclusively with game code, you should have a pretty good understanding of what it is doing under the hood; to reiterate, however, the game code is responsible for understanding the rules of the game being played. For example, it knows that *Capture the Flag* differs from pure deathmatch in that there are teams, different spawn points, two flags that must be captured and returned, and so on. The game code knows how each weapon behaves, and how players interact with the *Q3* world. In essence, the game code is ultimately responsible for the overall behavior of *Q3*.

The role of the game code extends beyond simply dealing with entities, rules, physics, and such. It is also responsible for dealing with the fundamentals of client/server technology. Because the actual core of making data connections from a *Q3* client to a *Q3* server is hidden away in the 3D engine code, the client/server logic is implemented in a much simpler layer—but it is no less important. In theory, it must be able to handle the connection and disconnection of players at any given time within the lifespan of the server. When new players connect to a *Q3* server, functions in game handle the addition of the player to the level, the assignment of various attributes to that player (such as name, model, skin, and so on), and the actual spawning of the player entity within the world. The game code also maintains functionality to allow clients to disconnect from the server, cleaning up any left-over data and properly removing their appropriate variables from *Q3*.

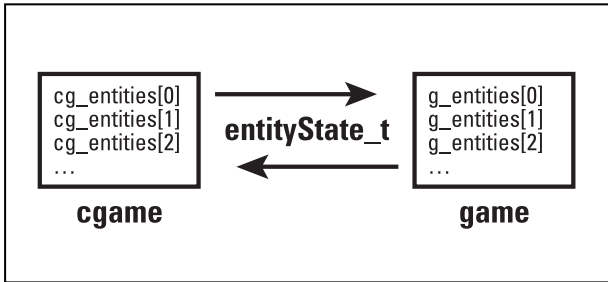
Because all the rules for client connection, disconnection, and game play reside within `game`, it makes sense that game code has the ultimate say in what happens to each player within the game. As you will see shortly, the `cgame` code assists the game code by handling as many of the *Q3* client-side events as possible, thus diminishing the amount of bandwidth required to play a game online. Because this separation of logic exists, so too does the possibility for a loss of synchronicity to occur; in other words, there is a possibility that the data on the client-side can begin to differ from data being seen on the server.

Ultimately, something must make the final decision in this situation, and it is the code in `game` that is assigned this responsibility. So, `game` code has the added capability to maintain consistency between what it sees is happening in a game, and all the clients that are participating on that server.

## The Main Event

The client's code responsibility is two-fold. First and foremost, it is responsible for handling interpolated states of the entities that are in the vicinity of the player. As discussed earlier, interpolation is the key to keeping bandwidth usage low between the client and server when playing a network game, because less data has to travel across the *pipe* (otherwise known as a network connection). Again, consider the example of a missile being fired by a player. In *Q3*, the `cgame` code has an understanding of how rocket physics behave, so if the server can send a simple tiny packet acknowledging the position and direction a rocket was fired, the `cgame` code can handle the movement of the rocket between server updates. In this manner, `cgame` assists in prediction.

`cgame` has another important role: spawning client-side events that cause visual and audible occurrences within the game. As you know, entities make up the backbone of all objects that exist in *Q3*; a player, a rocket, an explosion, and a box of ammo all represent entities. Entities exist on both `game` and `cgame`, and they maintain their synchronicity by passing only the most important data possible between the two modules. A good example of this is the struct `entityState_t`, a member of the `gentity_s` struct. Values in `entityState_t` are communicated between the `cgame` and `game` modules (see Figure 5.6), so that a certain amount of synchronicity can be maintained in two separate areas of code. Remember, because *Q3* is an online playable game, the

**Figure 5.6**

*Communicating  
entityState\_t between  
cgame and game*

cgame code and the game code can physically exist on two different computers, but they still must maintain data.

Because cgame has a list of entities that match the entities that exist in the game code, and because certain data is relayed between the two modules, there must be a way for the game code to alert the cgame code that things are happening to the entities. For example, sound effects need to be played when missiles whip past the player's head, special visual effects have to be applied to players who carry the Quad Damage powerup, icons need to be drawn up on the screen when a player makes an impressive attack, and so on; these are all examples of events, and they are handled in cgame.

Some events are shared between the cgame and game code. For example, when the game code sends an update to a client saying a certain player has died, it initiates the player's death event. On the client side of things, the cgame code creates the visual explosion and sends the gibs of player body parts flying across the screen. In this respect, both the game and cgame know what's going on: A player is dying.

There are also events that are exclusive to cgame only. For example, when a player in *Q3* presses the Tab key, the scoreboard appears on the screen. Here, the player can look at where he ranks, how many frags he's scored, what his current ping is, and possibly more info, based on the game. If the player is looking at the score-

## NOTE

*Ping* refers to the amount of time a small packet takes to complete a round trip from a client to a server. It gets its name from the sonar pings that submarines use to detect how close they are to objects underwater.

board (or not), there is no need to communicate that to the server-side code; it has no bearing on any other logic that the game code must make decisions around, and therefore, is a prime example of an event that is exclusive to `cgame` only.

In summary, `cgame` both assists in the prediction of events occurring around the player and handles client-side events by maintaining a necessarily small but important segment of data between entities on the client and server. You will soon learn that many changes to `game` will ultimately lead to changes in `cgame` (otherwise, `game` changes are fairly restrictive in what they can achieve). In Chapter 6, “Client Programming,” you’ll get the chance to dive into `cgame` in greater detail.

## Interfacing with the User

The *user interface* code establishes how you interact with the game. For example, the mouse and keyboard can be assigned certain functions in the game; the mouse, for example, can be used to aim and fire your weapon, and the keys of the keyboard can be used to move, switch weapons, and talk to other players. All these functions are established through *bindings* in the user-interface (`ui`) code. The `ui` files are responsible for building this interface, which allows you to set your particular configuration. It does this by first knowing the *controls* necessary to present this interface to you, and then by having an interface *layout* into which it can place these controls.

If you’ve ever used a form on the World Wide Web, you’re probably already familiar with certain types of controls, such as the text box into which you can type characters (such as your name), or the option button (a circle with a dot in it), which you can click to select or deselect an option. Additionally, you can use checkbox controls, drop-down menus, and so on, all of which enable you to enter data into the Web form.

Similarly, the *Q3* user interface features basic controls that let you alter game options and update your configuration. For example, one option button in the `ui` enables you to specify whether you want your weapons to auto-change when you pick them up (a yes/no question). Additionally, the `ui` features a textbox control into which you can enter your player’s name, and a slider control that allows you to specify how sensitive your mouse is to movement. In addition to playing with the



controls already included in the ui to build your own custom interface, you can take it a step farther and define your own controls!

An example of how the ui code can be modified to change the way the player interacts with *Q3* is demonstrated in the code differences between standard *Quake III Arena* and the Team Arena Expansion Pack. Take a look at Figure 5.7, which shows *Q3*'s in-game menu; note that the main menu is located in the center of the screen, with the choices listed down the middle. Then, check out Figure 5.8, which is the in-game interface from the Team Arena Expansion Pack. Wow! Even though it is the same game, the user-interface code allows the layout to completely change. The controls are laid across the top of the screen in a completely new fashion, and some new choices, such as About and Call Vote, are included.

This is just one example of what you can change by fiddling with the user-interface code. You'll toy around with this in Chapter 9, "UI Programming," which can be found on the CD-ROM.



**Figure 5.7** The standard *Quake III* in-game interface



Figure 5.8 The Team Arena Expansion Pack in-game interface

## Modifying Variables on the Fly

When a *Q3* game runs, certain variables that define the behavior of a given session are initialized with specific defaults. A good example is the `fraglimit` variable, which tells *Q3* what score players need to achieve in order to cycle to the next map. During the setup of a standard *Q3* server, the `fraglimit` variable is set to 20 by default; if no other options are changed, only 20 frags are required to cause the level to change. *Q3* allows that variable to be altered, either before setup or during the game in progress, through the use of console variables (*Cvars*). *Cvars* are simply variables that can be typed into the console to read and write variables directly into *Q3*. So if the host of a *Q3* game decides to change the maximum score required to change the level to 40, he could do so simply by typing

```
\fraglimit 40
```

In this case, the Cvar `fraglimit` is sent a new value to the game code (40), and the server-side logic can make the update appropriately. Alternately, one could set `fraglimit` before the game even launches by typing

```
quake3.exe +set fraglimit 40
```

As you can see, this is the same format as what you have been using to launch your modifications throughout this book; you are setting Cvars that tell *Q3* the directory from which to load game logic.

Cvars vary from variable to variable; some take effect right away, while others require a restart of the map currently being played.

Some Cvars are read-only, existing simply to retrieve data for the user; an example of this type of Cvar is `version`, which prints out the current build number of *Q3*. Some Cvars are module specific, such as

`cg_drawCrosshair`, which is a client-side Cvar that tells *Q3* which style of crosshair icon to draw in the center of the screen.

### TIP

To get a list of all the Cvars in *Q3*, press the `~` key to open the console, and type the following:

```
cvarlist
```

A flood of variable names (and their corresponding default values) should appear in the console. To scroll through them, use your Page Up and Page Down keys. To locate a specific variable, type the first few letters of it and press Tab; the console should try to complete the name for you, or present you with a list of names that come close.

## The *Quake* Virtual Machine

Now that you have a solid handle on `game`, `cgame`, `ui`, and the concept of console variables, one question remains: How do they all tie together? The answer is that the three modules communicate with each other through a common interface, the *Quake* Virtual Machine (QVM). This QVM then handles the relaying of communication

between the three modules back to the main *Q3* executable (the guts of the 3D engine).

## Getting the Best of Both (*Quake*) Worlds

To understand why communication is filtered through the QVM, you must examine decisions made during evolution of *Quake's* original code base. During the development of *Q3*, John Carmack, lead programmer at id, was gauging the importance of certain factors that were implemented in the previous incarnations of *Quake*. He liked having a simpler, lighter version of an interpreted C language that he could use to build the logic for his game, as was the case with his creation of *QuakeC*. The drawback to *QuakeC* was that it wasn't as fast as he would have liked.

In the creation of *Quake II*, Carmack felt that breaking the game logic out into a binary Win32 DLL was a great way to modularize the code, but that, too, had drawbacks. For one, it was not very secure. Any competent Win32 C programmer could devise a way to hook the DLL to other DLLs, which could ultimately cause some serious damage to the computer from which they were being executed. The other drawback involved portability, meaning a different DLL would have to be built for each platform on which *Quake II* was released.

The solution to these problems was to develop a way to modularize the code by allowing it to be compiled down into a new format—one that was secure, fast, and portable. Carmack announced that the solution would be implemented as a *Quake* Virtual Machine. That meant a programmer could develop and test a mod using the standard Win32 DLL format. Then, when it came time to release, he would compile the finished source down into a new native format, a QVM file, with a tool that Carmack developed called *q3asm*. This resultant QVM file would be able to run on any platform that *Q3* ran on—Windows, Linux, or Macintosh—without a need to be recompiled. In addition, QVM files had access only to their own private memory space, and would not be allowed to make any external system calls to the computer on which it ran; thus, the security risk was eliminated.

## Build Your Own QVM

Building a QVM is simple, so let's take a moment and build one now using the mod you created in Chapter 4 for implementing locational damage. Here's how:

1. Click your Windows Start button and choose Run.
2. In the Run dialog box, type **command** and press the Enter key on your keyboard.
3. The command prompt opens on your desktop (this is the same prompt you have been using to launch *Q3* with your mod). To change to the directory in which your mod source is stored, type the following line and press Enter:

```
cd \quake3\code
```

### NOTE

If Windows 2000 or Windows XP is your development platform, type **cmd** instead of **command** in the Run dialog box.

### TIP

If you have more than one hard disk, you may need to change to the correct drive before typing the command used for changing directories. For example, I store my mod source in my computer's C: drive, but my command prompt starts on the D: drive. To change drives, simply type the name of the drive you want and press Enter. For example, to change to the C: drive, type **C:** and press Enter.

4. From within the code directory (or folder), type **dir** and press Enter to view a list of files in that directory. Several files that end with **.BAT** should appear, such as **all.bat**, **cgame.bat**, **game.bat**, **q3\_ui.bat**, and so on. These are batch files, which you can execute by simply typing their names.
5. Type **all.bat** and press Enter. A stream of data should fill your command-prompt window, indicating that the various modules of *Q3*'s source are compiling. The difference now is that,

instead of Win32 DLLs being generated, you are creating *Q3* QVM files. When the process is complete, you should see the following:

```
0 total errors
code segment: 239092
data segment: 11712
lit segment: 20944
bss segment: 508824
instruction count: 78148
Writing to \quake3\baseq3\vm\q3_ui.qvm
Writing \quake3\baseq3\vm\q3_ui.map...
```

6. Using Windows Explorer, browse to the directory `\quake3\baseq3\vm`. In it, you should see some brand new QVM files, along with some additional MAP files.
7. Select the new QVM files, open the Edit menu, and choose Copy.
8. Browse to your MyMod folder.
9. In MyMod, locate the `qagamex86.dll` file and delete it. Don't worry! You can always recompile it later if you want to. This deletion is just to prove that you'll be running your Mod without the help of a DLL.
10. Still in MyMod, create a new folder called `vm`.
11. Open the `vm` folder, open the Edit menu, and choose Paste. The new QVM folders are pasted into the `vm` folder; you should now have a folder tree that reads something like `\quake3\MyMod\vm\`, with three files in it: `cgame.qvm`, `ui.qvm`, and `qagame.qvm`.
12. In the command-prompt window, type the following to return to the root `\quake3\` folder:  

```
cd \quake3\
```
13. In the root `quake3\` folder, type the following:  

```
quake3.exe +set fs_game MyMod +set sv_pure 0 +map q3dm1
```

Here, you are adding the `sv_pure` argument and setting it to 0; this launches *Q3* in a non-pure server mode. That means *Q3* is being told that some files necessary to run this game will differ from the standard *Q3* install, and can be located outside of a PK3 file.

**TIP**

**PK3 archives, which are used to house all the various files needed to run *Q3* properly, are in actuality nothing more than renamed ZIP files. If you want to peek at the contents of a PK3 file, just rename the file to have a .ZIP extension, and then open it in your favorite compression utility. Don't forget to rename it back to a PK3 when you're done snooping!**

After launching *Q3* and playing around a bit, you should notice that your changes from the locational damage mod are all still working as expected, yet no DLL exists for the changes to load from. That's because all the data for your mod is now wrapped nicely within the QVM files. If you had *Q3* running on another platform, such as Linux or Mac, you could copy those QVM files right over and be ready to go instantly—without making code changes or recompiling. The QVM is definitely a very convenient way of handling mod development, and I encourage you to use it!

## Summary

Data communication is the key that unlocks many of *Q3*'s secrets. Having spent some time on the fundamentals of how the various components interlock, you should now have a better understanding of why *Q3* was broken out into a modular format. You should also have a much clearer picture of each module's responsibilities, and what tasks each is best suited for.

You've spent a good deal of time analyzing game code up to this point in the book; now you'll turn your attention to the modules you haven't dealt with as much, beginning with learning more about `cgame` code and how it can be modified.



# CHAPTER 6

# CLIENT PROGRAMMING



**N**ow that you know the ins and outs of the three main segments of *Q3*'s code, let's start tinkering in the client code, something relatively untouched up to this point. As you will soon see, the client code is vital to mods. I hope to impress upon you the idea that the client and game code do indeed go hand in hand, and you will almost definitely be doing development in both projects from here on in. We're going to speed up very quickly, but trust me—this is the chapter where things really get exciting!

## Revisiting Weapons: Chain Lightning

You spent a good deal of time digesting weapon code in Chapters 2 and 3, but one weapon was left out, and for good reason: the beam weapon. Beam-weapon implementation in *Q3* tightly integrates the game and cgame code, so it would have been pointless for you to do any kind of modification to it that early. Now, however, you should be fully prepared to tackle the beam-weapon modification you are about to implement.

One of the coolest weapon modifications I have ever played around with is the chain-lightning gun, which is a simple modification to the standard lightning gun with one minor difference: When a chain-lightning gun is fired, a standard stream of lighting arcs toward the target. If any other targets are near, lightning then continues to arc toward the new target. The beam continues to hop from target to target, cascading damage across all of them. Let's take a look at what it will take to implement this change.

First, there is no function in *Q3* that matches the type of logic you want: the ability for an attacked target to test for other targets in lighting range. You will have to write that function into the game code. Secondly, there is no visual way in *Q3* to draw a beam of energy between two arbitrary points. This is another function you will have to

provide, and because it involves a visual effect, the code will be implemented as an event in the `cgame` code.

You'll write the logic to find new targets first, so start by opening up `g_weapon.c` and scrolling down to about line 660. You should find the start of the function

`Weapon_LightningFire`. You'll need a way to hold the all the potential targets that are found within the vicinity of the primary target, and an array seems the obvious choice. Above the definition for `Weapon_LightningFire`, add the following line of code:

```
static gentity_t    *chain_targets[MAX_CLIENTS];
```

It will be a static array, which means it will be visible only to this particular file to conserve memory (because it is already a global variable). Typically, you want to use as few global variables as possible, but in this case it's necessary. You can see that it's an array of type `gentity_t`, which is the same type of struct that holds all the entities in the game. After you have your array declared, go ahead and type in the code below. It's a hefty function, so I have numbered the lines so you can follow along more easily (remember, you don't have to type the numbers in).

```
1. void ChainLightning_Fire( gentity_t *ent, gentity_t *target ) {
2.     int          i, j, damage, currentTargetTotal;
3.     qboolean     isChainTarget;
4.     vec3_t       dir;
5.     gentity_t    *tent, *oldtarget;
6.     float        tent_dist, targetlength;
7.     trace_t      tr, targettr;
8.
9.     damage = 8 * s_quadFactor;
10.    currentTargetTotal = 1;
11.    chain_targets[0] = target;
```

## NOTE

The Team Arena Expansion pack for Q3 uses a new event, `EV_LIGHTNINGBOLT`, which creates a similar effect of lightning jumping from one point to the next. Because it's important that you understand everything that goes into the creation of an event, however, this section shows you how to custom-build a new event instead of just copying and pasting `EV_LIGHTNINGBOLT`.

```
12.
13.     while (1) {
14.         oldtarget = target; // save original target
15.         target = NULL;      // clear target pointer for reuse
16.
17.         for (i = 0; i < MAX_CLIENTS; i++) {
18.
19.             tent = &g_entities[i];
20.             isChainTarget = qfalse;
21.
22.             // is the entity in use?
23.             if ( !tent->inuse )
24.                 continue;
25.
26.             // is the entity the attacker?
27.             if ( tent == ent )
28.                 continue;
29.
30.             // is the entity the current primary target?
31.             for ( j = 0; j < currentTargetTotal; j++) {
32.                 if (tent == chain_targets[j]) {
33.                     isChainTarget = qtrue;
34.                     break;
35.                 }
36.             }
37.
38.             if (isChainTarget)
39.                 continue;
40.
41.             // is the entity too far for a standard light-
42.             ning beam to reach?
43.             VectorSubtract(oldtarget->r.currentOrigin, tent-
44.             >r.currentOrigin, dir);
45.             tent_dist = VectorLength(dir);
46.             if ( tent_dist > LIGHTNING_RANGE )
47.                 continue;
48.
49.             // does a valid trace occur between our primary
50.             target and the new target?
```

```

48.             trap_Trace( &tr, oldtarget->r.currentOrigin,
NULL, NULL, tent->r.currentOrigin, ENTITYNUM_NONE, MASK_SHOT );
49.             if ( tent != &g_entities[tr.entityNum] )
50.                 continue;
51.
52.             // all checks pass, we have a new target
53.             target = tent;
54.             targettr = tr;
55.             targetlength = tent_dist;
56.         }
57.
58.         if (!target)
59.             break;
60.
61.         if (!target->takedamage)
62.             break;
63.
64.         chain_targets[currentTargetTotal++] = target;
65.
66.         // Use tent to create the temporary lightning event
entity
67.         tent = G_TempEntity( targettr.endpos, EV_MISSILE_HIT );
68.         tent->s.otherEntityNum = target->s.number;
69.         tent->s.eventParm = DirToByte( targettr.plane.normal );
70.         tent->s.weapon = WP_LIGHTNING;
71.
72.         // Deal out the damage
73.         G_Damage( target, ent, ent, dir, targettr.endpos, dam-
age, 0, MOD_LIGHTNING );
74.
75.         // Signal for the lightning event
76.         tent = G_TempEntity( oldtarget->r.currentOrigin,
EV_LIGHTNINGARC );
77.
78.         // Set the destination of the arc for the client event
to pickup
79.         VectorCopy( target->r.currentOrigin, tent->s.origin2 );
80.     }
81. }

```

This is quite a function, so let's break it down part by part. After getting all your variables declared, the first thing you do is set the damage parameter (line 9), or the amount of damage dealt with each arc of lightning that contacts a target. Because the original `Weapon_LightningFire` sets the damage to 8 multiplied by `s_quadFactor`, you can do the same here. You also want to create a variable to hold the total number of targets being hit by chain lightning; you can do that with `currentTargetTotal` (line 10). Start by setting it to 1, because you already have one target when this function is called. As well, you use the first index of the static `gentity_t` array to hold the current target being hit—the very same target variable that is passed into this function (line 11).

You may have seen `while(1)` before, littered throughout the *Q3* code. This is a tricky bit of code to which you should pay careful attention. Typically, a `while` loop continues to execute as long as the logic in parentheses evaluates to true. Because the only bit of code in parentheses is the number 1, you're looking at an infinite loop—a chunk of code that repeats forever. This is not a good thing, unless you enjoy locking up your computer. In this instance, the infinite loop is handled carefully, so that it is broken when the proper indicators are flagged. Bear in mind that this function, `ChainLightning_Fire`, is constructed in a manner to continually look for new targets once an initial target is hit. For this, an infinite loop is exactly what you want: constant updates for new targets. Just know that when you reach this code, the loop will begin, and it is your job to stop it.

As the loop begins (line 13), you assign `oldtarget` a value equal to the current target, and then clear the target variable by setting it to `NULL`. Next, you begin to loop through all the clients in the game; the easiest way to achieve this is to loop from 0 to the global variable `MAX_CLIENTS` (line 17). In this loop, you assign a temporary variable called `tent` equal to the entity found in the global entity array, `g_entities`. Note the usage of the address-of symbol (`&`), which means `tent` is physically that entity! If you change `tent` now, you'll be changing the actual entity in the game!

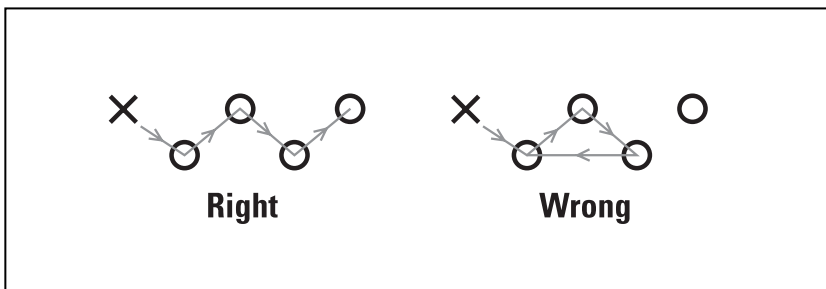
## NOTE

The functions that breathe life into the chain lightning gun featured in this chapter are based on code created for the *MaxCarnage* mod, created by Chris Hilton.

Once you have your temporary entity, you need to perform some checks on it to see if it is a valid player entity capable of being hit by an arc of chain lightning. The first check, `if (!tent->inuse)`, checks whether the entity is not currently being used by *Q3* (line 23). If it isn't, the `continue` keyword causes control to be passed to the next iteration of the loop, skipping the remaining checks. If it passes the check, however, it then looks to see if the entity equals the player firing the gun, shown with `if (tent == ent)`. You certainly do not want the chain lightning to arc back to the player firing the gun, so if this check is `true` (line 27), the loop skips to the next entity as well.

Next, a check is made to see if the temporary entity equals any current target that is already being hit by chain lightning. The idea is that you will keep track of a list of targets already being hit by lightning, so that they can no longer be a valid target searched for at the end of the chain. If you're confused, take a look at Figure 6.1. The chain lightning path on the left is correct; it leaps from target to target, never hitting the same player twice. The chain on the right, however, is incorrect, because the first target is also the third target in the chain, a behavior you want to avoid.

Remember that you're keeping a list of all players being hit by assigning them a spot in `chain_targets`. Because this is your first trip through the loop, the array only has one index, 0, which points to the original target. Soon you will see how this array grows, but for now, execute the loop starting at 0 and ending one less than the value of `currentTargetTotal` (line 31). Within the loop, check whether `tent` is equal to the current entity in `chain_targets` (at that index of the loop). If it is, set `isChainTarget` to `true` (actually, to `qtrue`, because it is



**Figure 6.1** The correct (left) and incorrect (right) execution of chain lightning

qboolean), and break out of the list of `chain_targets` with `break`. On the very next line, look at the value of `isChainTarget` (line 38); if it is true, it means the temporary entity is already a target being hit, so move to the next entity with the `continue` keyword.

The next bit requires a little vector math. Don't worry! You've done fine working with vectors before; this is no different. You will want to create a distance vector that is equal to the distance between the last target in the chain (remember, you set it to `oldtarget`), and the current temporary target, `tent`. Do that with a simple call to `VectorSubtract`:

```
VectorSubtract(oldtarget->r.currentOrigin, tent->r.currentOrigin, dir);
```

Here, you are subtracting the distance between `oldtarget` and `tent`, and saving the results in `dir`. Then, you convert that vector into a scalar of distance by passing it to the `VectorLength` function:

```
tent_dist = VectorLength(dir);
```

The new distance measurement is saved in `tent_dist` (line 43). Then, test to see if `tent_dist` is greater than the lightning gun's standard range, which is saved in a global variable called `LIGHTNING_RANGE`. If it is, this target is no good, so move to the next entity with good old `continue`.

## Trace Your Path

The next test the temporary entity must pass is whether a valid lightning bolt could hit it, based on other entities or obstructions that may lie within the level between the two targets. You certainly don't want to hit a target that's on the other side of a wall, because lightning can't pass through walls in *Q3*. You perform this check with a call to `trap_Trace`. As you have seen time and time again, `trap_Trace` is a system call function:

```
void trap_Trace( trace_t *results, const vec3_t start, const vec3_t mins, const vec3_t maxs, const vec3_t end, int passEntityNum, int contentmask )
```

`trap_Trace` is a function of void type, which means it returns nothing. The reason it returns nothing is that it actually assigns the value of what it finds to the first input parameter, `results`, which you can see is

passed in as a pointer. `results` is a new type of variable, `trace_t`, which you will examine in a second. The second parameter is a vector at which to start the trace. The third parameter is a `mins` value, followed by a fourth parameter, a `maxs` value. This should spark some memory of bounding boxes, which were discussed in Chapter 4. This means you're not restricted to just tracing a line from one target to another; you could trace the entire width of a bounding box extended along that line. Pretty cool, eh? (Those guys at id aren't game developers for nothing!) The fifth parameter is an ending vector, where your trace will stop. The sixth parameter is slightly obscure; it references an integer called `passEntityNum`, which you can use to ignore a specific entity. In this case, you ignore nobody, and can do that by passing in a global variable called `ENTITYNUM_NONE`. The final parameter is an integer that represents a content mask. In a nutshell, you can hide or "mask" your trace from certain entity types, meaning if the trace hits an entity found within the mask, it will stop tracing. `MASK_SHOT` is a global variable representing solid walls and other players, living or dead. Table 6.1 shows the current masks in *Q3* and their definitions.

**Table 6.1 Content Mask Flags**

<b>Name</b>	<b>Types Included in Mask</b>
<code>MASK_ALL</code>	Everything and anything
<code>MASK_SOLID</code>	Solid walls, floors, and roofs
<code>MASK_PLAYERSOLID</code>	Everything in <code>MASK_SOLID</code> , plus living players or players' bounding boxes
<code>MASK_DEADSSOLID</code>	Everything in <code>MASK_SOLID</code> , plus players' bounding boxes
<code>MASK_WATER</code>	Water, lava, or slime
<code>MASK_OPAQUE</code>	Everything in <code>MASK_SOLID</code> , plus slime or lava
<code>MASK_SHOT</code>	Everything in <code>MASK_SOLID</code> , plus players or dead bodies



When all the parameters are passed into `trap_Trace`, if a valid entity is hit, it is returned in a `trace_t` struct, which is defined as follows:

```
typedef struct {
    qboolean    allsolid;        // if true, plane is not valid
    qboolean    startsolid;     // if true, the initial point was in a
    solid area
    float       fraction;       // time completed, 1.0 = didn't hit any-
    thing
    vec3_t      endpos;         // final position
    cplane_t    plane;         // surface normal at impact, transformed
    to world space
    int         surfaceFlags;   // surface hit
    int         contents;       // contents on other side of surface hit
    int         entityNum;      // entity the contacted surface is a part of
} trace_t;
```

Ah, if only every function in *Q3*'s source was as gracefully documented as this! As you can see, there are a number of members of this struct, but the one you want to deal with is the last one, an integer called `entityNum`. Because you worked with `entityNum` way back in Chapter 3, you may recall that all entities are numbered, and can be referred to by that number. This is how you check to see if the `trap_Trace`'s returned `entityNum` belongs to that of your temporary entity (lines 48 through 50).

This shows that you trace a line from `oldtarget` to the temporary entity `tent`, ignoring nobody and using the `MASK_SHOT` mask. Notice that you also pass in `NULL` values for `mins` and `maxs`, which creates a basic line. If any entity was hit, its `entityNum` is returned in the `tr` variable. If `tent` is not the entity that was hit in the trace, it's back to the loop you must go, to keep checking for a valid target. Otherwise, congratulations! You've found a valid player entity that will become the next target in the chain. Make it so by assigning `target` equal to `tent` (line 53). You will also want to hang onto the trace variable, so assign it to `targettr` (line 54). Finally, that distance measured between the target and the new target will also be of value, so assign it to `targetlength` (line 55).

Once out of the loop of all the clients (`MAX_CLIENTS`), you must perform some sanity checks. There is the definite possibility of looping

through all the clients and still not having a valid target. So, because you set `target = NULL` when the loop began, check to see if it is still `NULL` (line 58). If it is, end your loop and start the process all over again with `break`. You will also want to break out of the loop if the target is not able to take damage (line 61).

To validate your new target, add it to the `chain_targets` array, incrementing the `currentTargetTotal` variable in the process (line 64). This will ensure that the target can never be retargeted by the same chain. Because you have a valid target, you need to create the chain lightning to hit it.

## TIP

**Just to be sure you're on the right page, the differences between `break` and `continue` in a loop are as follows:** The `continue` keyword passes control of the function to the next iteration of the loop it is currently within. The `break` statement, on the other hand, immediately steps out of the current loop, regardless of what index it's at.

## Be Like Zeus

Now it's time to get a little like Zeus and start working the lightning bolts. You already have the `tent` variable set aside to be an entity; you are free to reassign its role from being a possible temporary player entity to being a real temporary event entity. You start by calling a client event that is already defined in *Q3*, `EV_MISSILE_HIT` (lines 67 through 70). This causes a visual effect of a small lightning blast on the enemy (the arc will follow shortly). Then, you deal out the damage with your good friend `G_Damage` (line 73). Once those chores are out of the way, it's time to call your new client event, the one that will create the visual effect of an arc of lightning jumping from your target to a new target.

First, you reuse `tent` a second time and pass it to `G_TempEntity` (line 76), which creates your new client event, located at `oldtarget`'s position. (Recall that the `oldtarget` variable will equal the value of the last target in the chain at the time of execution.) The type of event will be `EV_LIGHTNINGARC`. Then, you copy the vector of the new target (currently assigned to the `target` variable) into the temporary entity's `s.origin2` value (line 78). If that's a bit confusing, here's the

plain-English translation: You have created an entity that will start an arc of chain lighting at the current target being hit, and will end at the newly found target's center point.

I know you're itching to create the new `EV_LIGHTNINGARC` event (I am too!), but you have one more quick change to add before you leave the game code. You need to update the original lightning gun to call the new `ChainLightning_Fire` function. Scroll down into the `Weapon_LightningFire` function, and find the line that deals the damage out via `G_Damage` (it should be about line 733 in Visual Studio now). Add the call to `ChainLightning_Fire` right after `G_Damage`, so the lines together read as follows:

```
G_Damage( traceEnt, ent, ent, forward, tr.endpos,
          damage, 0, MOD_LIGHTNING);
          ChainLightning_Fire( ent, traceEnt ); // initial
target hit, now find more targets
```

You call `ChainLightning_Fire` by passing in the attacker, held in `ent`, and the current traced-to target for the standard lightning gun, held in `traceEnt` (yep, the original lightning gun also uses `trap_Trace!`). Give yourself a pat on the back; you've done everything in the game code that will support the new chain-lightning gun. Now it's time to create a little magic in the client code.

## Creating Client Events

As discussed in Chapter 5, the client code has a basic understanding of what is going on in the game, but for the most part, it is a separate beast from the game server code altogether. Both `cgame` and `game` are responsible for their own upkeep, but every once in awhile they do need to communicate. One such occasion is when the server-side code alerts clients that an event is to occur—something visual or audible that each player will experience on his screen. Because the server-side game code is far too busy handling more complicated issues, it hands the task off by passing a single event to the players. When the client code in each player's *Q3* game receives this event, it takes over and creates the intended effect, be it a blast of shotgun shells, a sound effect of a missile as it flies past your head, or the explosion of a player into bloody gibs. Like all of these occurrences, your chain

lightning mod will have a new event assigned to it, which will create the visual effect of arcs of lightning leaping from target to target.

## Enumerating an Event

Start by opening `bg_public.h` and scrolling to about line 430. Here, you see the ending to an enum definition, called `entity_event_t`. An *enum* is very similar to a struct, except that it contains only variables that equal a certain integer. Typically, the integers start at 0 and count up automatically, but you can assign them a different start point if you wish. Most of the enums in *Q3*'s code start at 0 and go up from there. The enum `entity_event_t` holds a list of variables describing events that are known to both the game code and client code. The game code knows them only by their enum variable name. The client code, on the other hand, knows the dirty secret behind each variable in the `entity_event_t` enum. Add your new `EV_LIGHTNINGARC` event at the very end of the enum, so that the final few lines read like this:

```

        EV_TAUNT_GUARDBASE,
        EV_TAUNT_PATROL,

        EV_LIGHTNINGARC // arc of lightning
    } entity_event_t;
```

Excellent. Next, you want to prototype or “declare” the function that will handle your new arc of lightning event. Place the prototype of the function near the bottom of `cg_local.h`, right after `CG_Bleed` on line 1395:

```

void CG_Bleed( vec3_t origin, int entityNum );

void CG_LightningArc( vec3_t start, vec3_t end ); // our new event!
```

Here, you have set the stage for the as-yet-to-be-written `CG_LightningArc` function, which is of void type and takes two input parameters: a vector to start, and a vector to end. Let's go ahead and write this function next.

Open up `cg_effects.c`. This file is chock full of client-side effects; performing a quick scan down the list reveals functions that gib a player into a bloody mess of flying chunks, a function to show the player

bleeding, and a function to show the player teleporting. Scroll all the way to the bottom and add your new event function, which reads as follows:

```

/*
=====
CG_LightningArc

Generates an arc of lightning between
two arbitrary vectors
=====
*/
void CG_LightningArc( vec3_t start, vec3_t end ) {
    refEntity_t      arc;

    memset( &arc, 0, sizeof( arc ) );
    arc.reType = RT_LIGHTNING;
    arc.customShader = cgs.media.lightningShader;

    VectorCopy( start, arc.origin );
    VectorCopy( end, arc.oldorigin );

    trap_R_AddRefEntityToScene( &arc );
}

```

Kind of a short and sweet function, isn't it? As you will see, great things do come in small packages. In this function, you will start by defining a temporary variable called *arc*, which is of a new type, *refEntity\_t*. The *refEntity\_t* struct is declared in the file *tr\_types.h* on line 54:

```

typedef struct {
    refEntityType_t    reType;
    int                renderfx;

    qhandle_t         hModel;           // opaque type outside refresh

    // most recent data
    vec3_t            lightingOrigin;    // so multi-part models can be
lit identically (RF_LIGHTNING_ORIGIN)
    float             shadowPlane;      // projection shadows go here,
stencils go slightly lower

```

```

    vec3_t      axis[3];           // rotation vectors
    qboolean    nonNormalizedAxes; // axis are not normalized, i.e.
they have scale
    float       origin[3];        // also used as MODEL_BEAM's
"from"
    int         frame;           // also used as MODEL_BEAM's diam-
eter

    // previous data for frame interpolation
    float       oldorigin[3];     // also used as MODEL_BEAM's "to"
    int         oldframe;
    float       backlerp;         // 0.0 = current, 1.0 = old

    // texturing
    int         skinNum;          // inline skin index
    qhandle_t   customSkin;       // NULL for default skin
    qhandle_t   customShader;     // use one image for the entire
thing

    // misc
    byte        shaderRGBA[4];    // colors used by rgbgen entity
shaders
    float       shaderTexCoord[2]; // texture coordinates used by
tcMod entity modifiers
    float       shaderTime;       // subtracted from refdef time to
control effect start times

    // extra sprite information
    float       radius;
    float       rotation;
} refEntity_t;

```

There's quite a lot of obscure information going on in there, but I think you can determine what some of those values mean by now. Specifically, you will use the members `reType`, `customShader`, `origin`, and `oldorigin`. You'll be happy to know that `reType` is of yet another new variable type, an enum of type `refEntityType_t`, which if you'll recall simply maps to an integer value. Luckily for you, the declaration of `refEntityType_t` is directly above the declaration of `refEntity_t`, so if you're curious, you won't have to scroll far to see all the values. The one value that's of importance to you is `RT_LIGHTNING`.

After you create your arc variable, the first thing you want to perform is a memory-clear of the variable, done with a call to `memset`. This is simply a good practice to get into, because initializing complex variables in C often forces the program to pull memory from random places in the stack, sometimes pulling from memory that already had old, unused garbage data in it. Passing the new variable into `memset` with a value of 0 ensures that the value is cleared and ready to go.

Next, set the `reType` member of `arc` to `RT_LIGHTNING`, and set the `customShader` member to `cgs.media.lightningShader`. This is so that the new arc of lightning will look and behave like a standard bolt of lightning fired from a weapon.

Then use `VectorCopy` to copy the passed in vectors `start` and `end` to the arc's appropriate counterparts: `origin`, and `oldorigin`, respectively. Finally, signal to `Q3` to add this new chain-lighting event to the game by passing the `arc` variable to `trap_R_AddRefEntityToScene`. This new function is another of the now-familiar system calls that simply pass the data to a direct line within `Q3`'s executable file, causing the event to be placed in the game.

## NOTE

After you complete the modification of the chain-lighting gun, I'll give you a closer look at shaders, what they represent, and how to make a new one.

## The Communication of `entityState_t`

The final change you must make to your code in order to allow the new `CG_LightningArc` event to fire is to hand it off to `Q3`'s main event handler. Open `cg_event.c` and scroll down to about line 445. Here, you should see the function `CG_EntityEvent`. This is where every event that can be sent to `Q3` is handed off to each appropriate function. If an event is created, it will ultimately land in the hands of `CG_EntityEvent`, so this is definitely where you want to make your call. As you can see by peering down the pages of code, there are a LOT of events to take care of. Heck, what's one more? Let's add your new event in here now.

Scroll down to about line 834, right after the event handler for EV\_GRENADE\_BOUNCE, and add the following code:

```
case EV_LIGHTNINGARC:
    CG_LightningArc( position, es->origin2 );
    break;
```

That was nothing! I didn't even break a sweat typing that, how about you? Here, the list of events is handled in a simple switch block. And as all good C programmers know, to add a new bit of code to a switch, you use the case keyword. Here, you add a case for EV\_LIGHTNINGARC, which will call the new function CG\_LightningArc, passing in two parameters. The first is position, which is a vector passed into CG\_EntityEvent, representing the location of the event, in its starting place. The second parameter is es->origin2. es is a variable set at the top of CG\_EntityEvent, which points to cent->currentState. So what is cent, you ask? cent is actually the other variable passed into CG\_EntityEvent, and it is of type centity\_t. Let's take a look at what that type of variable is.

The struct centity\_t is declared at line 149 in cg\_local.h:

```
typedef struct centity_s {
    entityState_t    currentState;           // from cg.frame
    entityState_t    nextState;             // from cg.nextFrame, if
available
    qboolean         interpolate;           // true if next is valid
to interpolate to
    qboolean         currentValid;         // true if cg.frame
holds this entity

    int              muzzleFlashTime; // move to playerEntity?
    Int              previousEvent;
    Int              teleportFlag;

    Int              trailTime;           // so missile trails can
handle dropped initial packets
    Int              dustTrailTime;
    Int              miscTime;
```



```

    Int                snapShotTime;    // last time this entity
was found in a snapshot

    playerEntity_t    pe;

    int                errorTime;       // decay the error from
this time
    vec3_t             errorOrigin;
    vec3_t             errorAngles;

    qboolean           extrapolated;    // false if origin /
angles is an interpolation
    vec3_t             rawOrigin;
    vec3_t             rawAngles;

    vec3_t             beamEnd;

    // exact interpolated position of entity on this frame
    vec3_t             lerpOrigin;
    vec3_t             lerpAngles;
} centity_t;

```

Interesting. This `centity_t` struct does bear a few similarities to a struct with which you are already familiar: `gentity_t`. Could it be that *c* stands for *client* while *g* stands for *game*? Absolutely. The `centity_t` struct is used for client-side maintenance of entities. To find out where `cent` is really coming from, let's trace the steps back from `CG_EntityEvent`.

`CG_EntityEvent` is called at the end of `CG_CheckEvents`, a function that also has `cent` passed into it (so control of `cent` is passed directly from `CG_CheckEvents` to `CG_EntityEvent`—the same variable all the way through). `CG_CheckEvents` is called at the end of `CG_TransitionEntity`, which also receives `cent` as an input parameter, passing it directly through. (Are you beginning to get the idea that `cent` is important?) Finally, `CG_TransitionEntity` can be traced back to a call from `CG_TransitionSnapshot` in the following code:

```

for ( i = 0 ; i < cg.snap->numEntities ; i++ ) {
    cent = &cg_entities[ cg.snap->entities[ i ].number ];
    CG_TransitionEntity( cent );
}

```

From this code, it looks as though `cent` is a variable whose value points to a specific index of a global array called `cg_entities`. Whoa, *dèjà vu!* This is almost exactly what happens in the game code, when you loop over entities by assigning a temporary variable to a given index in the array `g_entities`! By now, you should have probably guessed the result: `cent` is a client-side entity that has a partner on the server side, and it maintains its state from the two modules by the passing of `entityState_t` to `cent->currentState`. Remember when you used this line of code in game, near the end of your `ChainLightning_Fire` function?

```
VectorCopy( target->r.currentOrigin, tent->s.origin2 );
```

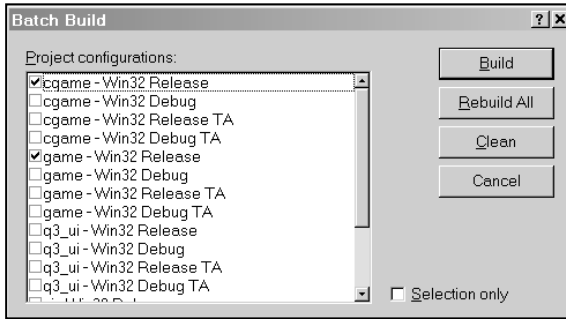
Well, take another look at your `CG_LightningArc` function call:

```
CG_LightningArc( position, es->origin2 );
```

See something familiar? When you performed the `VectorCopy`, you took a copy of the location of the newest target being hit by chain lightning and passed it into your temporary entity's `s.origin2` variable. `s`, if you recall, is the `entityState_t` struct of `gentity_t`. And, if your memory is very good, you'll also remember that `entityState_t`'s values are communicated from the server to the client. I have already shown that the client-side variable `es` points to a `cent->currentState` variable, and `currentState` is also an `entityState_t` struct. Therefore, the ending position you're passing to `CG_LightningArc`, which is `es->origin2`, is actually a client to server—communicated `entityState_t` value, coming from the original temporary entity `tent` you created in `ChainLightning_Fire`!

Yikes! That's some pretty technical stuff. If you've made it this far and can still see straight, good for you. Take a deep breath and let's press on. With the final addition of the `CG_LightningArc` event to `CG_EntityEvent`, you have successfully implemented the client-side event of your new chain-lightning gun. Now is

**NOTE**  
 The specifics of communicating the `entityState_t` structs from server to client are even more complicated than I have shown here. (I discussed it briefly in the previous chapter.) It's important only that you understand the semantics behind the creation of a temporary entity, signaling an event, and accessing its corresponding value on the client.



**Figure 6.2** The Batch Build dialog box in VC++ with game *and* cgame selected

the time of judgment. Go ahead and compile both the game code and cgame code. Remember, you can right-click one of the projects in your FileView tab and select Set As Active Project to prepare for a build of a specific DLL. Or, you can open the Build menu in Visual Studio, select Batch Build, and put checkmarks next to cgame—Win32 Release and game—Win32 Release only, as shown in Figure 6.2. Then, click the Build button, and VC++ does its thing.

## Chain Lightning Lives

After compiling, verify that you indeed have a cgamex86.dll and a qagamex86.dll in your MyMod folder. If you compiled to a custom directory, be sure to move your new DLLs over to /quake3/MyMod/. Then, test your new DLLs with the following:

```
quake3.exe +set fs_game MyMod +set sv_pure 0 +map q3tourney2
```

Don't forget to set sv\_pure to 0 (you have not placed your files in a PK3 archive yet). If all goes well, you should be able to storm through a map with several bots and see arcs of chain lightning zig-zagging across it, as pictured in Figure 6.3.

Forgive me for being excited, but that's some incredibly cool stuff! Not only have you modified the lightning gun's behavior in the game code, you've create new visual effects of chain lightning in the client code. Hopefully, you can see why I encouraged you to tackle this chapter at the start. Harnessing the client-side code via events is a powerful



**Figure 6.3** Arcs of chain lightning hitting multiple bots

way to create mods, because the changes you make are visual and, therefore, more noticeable. As you explore the client code in greater detail, you'll see that you have an awesome array of programming tools at your disposal that allow you to play with visuals and audio. The next part of this chapter is dedicated to one of those tools, the shader.

## Working with Shaders

In order to achieve some heavy-hitting visuals in *Q3*, the developers back at id devised a streamlined manner to implement special effects. They achieved this by creating a uniform scripting language that handled the way surfaces were represented in the 3D world. A common set of functionality was applied to these surfaces through these scripts with various attributes, allowing the developer to quickly bring a texture to life within the game, making it glow, vibrate, pulse, or more. These scripts are known as *shaders*.

Most of the shaders are hidden from view, packed away nicely in the PK3 archives that contain all the other important data for *Q3*. If you extract all the files from a PK3 archive, however, you find the shaders in the `/scripts/` directory. A code snippet of this scripting language looks like so:

```
powerups/quad
{
    deformVertexes wave 100 sin 3 0 0 0
    {
        map textures/effects/quadmap2.tga
        blendfunc GL_ONE GL_ONE
        tcGen environment
        tcmmod rotate 30
        tcmmod scroll 1 .1
    }
}
```

Here is the definition for the quad powerup shader, which wraps around the player when he picks up the quad-damage powerup. `deformVertexes` is a keyword that tells *Q3* that this shader will inevitably change the shape of the surface to which it is applied. Some shaders handle only visual coloring and glow-style changes, while others physically manipulate the shape of the surface, as this example will. The `wave` keyword indicates that this shader will attempt to handle the surface as water, while the `sin` keyword indicates the type of constant math function applied to the manipulation of the texture. In case you didn't know, `sin` stands for *sine*, a trigonometric function that, when graphed, produces an oscillating wave form, bearing a striking resemblance to water. Thus, `sin` is the perfect function to apply to a texture that needs to exhibit wave-like properties.

The value of 3 indicates how many units “above” the surface the shader will be applied. Because the quad-damage shell effect sits outside the player model, a positive value reflects this position. (0 would denote the surface itself, while a negative number would apply the shader effect below the surface.) The three trailing zeros are additional parameters that can be used to affect the style of shader; in this case, no more surface changes are required. Additional parameters exist with the curly brackets that tell *Q3* such things as what image file will be used in the shader and how the effect will blend, move, rotate, and scroll through the image file used.

**NOTE**

Unfortunately, there is a lot of information regarding the creation and manipulation of shaders—enough to fill an entire chapter (or more) in this book. Here, I'll focus on getting a new shader up and running; then, near the end of the book, I'll provide you with some online resources to further your knowledge of shaders.

## A Shady Modification: Armor Regeneration

This shader scripting language is quite powerful and easy to use. Because you're working with a modification in progress, let's see what it will take to apply a shader to it. I think a really cool effect would be to make the chain-lightning gun allow the player to regenerate armor when he successfully creates a chain between two or more targets. And, to signify that armor is being regenerated, you can apply a glowing shell to the player, much like the quad-damage effect.

Start by opening `bg_public.h` and heading to line 247. Here the enum `powerup_t` is defined. The values of `powerup_t` map to flags that represent the various powerups that are in *Q3*; you may recognize `PW_FLIGHT` from Chapter 4. Right after `PW_FLIGHT`, add your new flag:

```
PW_REGEN,  
PW_FLIGHT,
```

```
PW_ARMORDRAIN, // our new  
visual shell effect
```

Here, you add `PW_ARMORDRAIN`, a flag that will represent the effect of armor being regenerated on the player.

**NOTE**

Be careful when you add powerups to this `powerup_t` struct. As a C-style comment notes near the top of the declaration, there can only be 16 powerups at one time. I'll let you in on a secret about this restriction in Chapter 9, "UI Programming," on the CD-ROM, but for now, if you need to add new variables, simply rename old ones and use them.

Next, you will need to tell the `cgame` code that it can expect a new shader to be loaded into memory when *Q3* initializes a new game session. Creating a new handle to a shader through the use of another *Q3*-specific datatype, `qhandle_t`, does this. Open `cg_local.h` and scroll down to about line 744, where you should see the declaration of the `quadShader` variable. Add a new line below that, and insert your new shader like so:

```
// powerup shaders
    qhandle_t    quadShader;
    qhandle_t    armorDrainShader;    // new handle for armor drain
```

This will allow *Q3* to set aside memory for a new shader when it comes time to load it into the game and give the code a point of reference to that variable. Now you need to load the shader into the game when a new client initializes. Do this using `CG_RegisterGraphics`, which sits in `cg_main.c` at line 789.

In brief, the function `CG_RegisterGraphics` loads once when a client first sets up after loading *Q3*. It churns through all the various texture images in the *Q3* game, and starts assigning them to shader variables that will be referenced later, as the game is played. This is known as *caching* data; a cache is frequently used in all kinds of applications aside from games. Thanks to a cache, when it comes time to actually make use of some physical data outside the executable, the application does not have to take extra time to access the hard disk, load the file, and then deal with it appropriately. Disk access is always slower than memory access, so using a cache is a smart way to handle external data.

`CG_RegisterGraphics` is where you will add a cached referenced to your new armor-drain shader. Scroll down to about line 860, where a bunch of powerup shaders are being cached. After the `quadShader` variable is set, add one for your new shader like this:

```
// powerup shaders
    cgs.media.quadShader = trap_R_RegisterShader("powerups/quad" );
    cgs.media.armorDrainShader =
trap_R_RegisterShader("powerups/armordrain" );
```

As you can see, a global struct called `cgs`, which contains a member `media`, now has a third member in its hierarchy, `armorDrainShader`. The value of the `armorDrainShader` property is assigned by making a call to

another system call function named `trap_R_RegisterShader`, passing in the path to shader script.

The shader script does not exist yet, but you will write it shortly. The setup of the shader is complete at this point, so the next task is to actually apply it to the player currently regenerating armor.

## NOTE

The global variable `cgs` is of data type `cgs_t`, which is declared on line 969 of `cg_local.h`. The `qhandle_t` you added was made a member of a struct called `cgMedia_t`, which itself is used as a data type of the variable `media`, which in turn is a member of `cgs`.

## Making the Player Glow

To apply the effect to the player, the shader you have created will need to be added to the game at the appropriate time. Luckily, this particular shader works exactly as the other powerups do, so there is no better way to do this than by looking at the current powerup function, `CG_AddRefEntityWithPowerups`. You will find this function on line 2118 of `cg_players.c`:

```
void CG_AddRefEntityWithPowerups( refEntity_t *ent, entityState_t
*state, int team ) {

    if ( state->powerups & ( 1 << PW_INVIS ) ) {
        ent->customShader = cgs.media.invisShader;
        trap_R_AddRefEntityToScene( ent );
    } else {
        trap_R_AddRefEntityToScene( ent );

        if ( state->powerups & ( 1 << PW_QUAD ) )
        {
            if (team == TEAM_RED)
                ent->customShader = cgs.media.redQuadShader;
            else
                ent->customShader = cgs.media.quadShader;
            trap_R_AddRefEntityToScene( ent );
        }

        if ( state->powerups & ( 1 << PW_REGEN ) ) {
```



```

        if ( ( ( cg.time / 100 ) % 10 ) == 1 ) {
            ent->customShader = cgs.media.regenShader;
            trap_R_AddRefEntityToScene( ent );
        }
    }
    if ( state->powerups & ( 1 << PW_BATTLESUIT ) ) {
        ent->customShader = cgs.media.battleSuitShader;
        trap_R_AddRefEntityToScene( ent );
    }
}
}
}

```

This function requires the input of a `refEntity_t`, an `entityState_t`, and a team variable. Here, `ent` refers to the client-side version of the player entity, while `state` refers to the currently communicated state of the entity in question. The various chunks of code in this function look at the state of the powerups in the `entityState_t` struct; if they include a flag referencing a specific powerup, then the client-side entity's `customShader` member is set to the appropriate shader variable. After the new shader is set, it is physically added to the visuals in the game with `trap_R_AddRefEntityToScene`, yet another system call that ties directly to the 3D rendering code.

After the code that adds the quad shader, add the following code segment, so that it looks like this:

```

        ent->customShader = cgs.media.quadShader;
        trap_R_AddRefEntityToScene( ent );
    }

    // check for the armor drain activity of chain lighting gun
    // if it is a powerup found on a player, draw the shell effect
    if ( state->powerups & ( 1 << PW_ARMORDRAIN ) )
    {
        ent->customShader = cgs.media.armorDrainShader;
        trap_R_AddRefEntityToScene( ent );
    }
}
}
}

```

I shouldn't be surprising you at this point. You make a simple check to see if the new powerup flag, `PW_ARMORDRAIN`, is presently found on the `entityState_t` passed in. If so, the `armorDrainShader` is assigned to

the entity's `customShader` member, and the updated entity is sent to `trap_R_AddRefEntityToScene`.

Of course, when a player is being drained of armor, there should be some visual effect to indicate this both to that player and to the other players in the game. Because the player never sees himself, but can see his weapon model bobbing and weaving as he runs through a level, let's apply the visual effect to the weapon as well as to the player.

Open the file `cg_weapons.c` and scroll down to about line 1168. You should see the definition of the function `CG_AddWeaponWithPowerups`:

```
/*
=====
CG_AddWeaponWithPowerups
=====
*/
static void CG_AddWeaponWithPowerups( refEntity_t *gun, int powerups )
{
    // add powerup effects
    if ( powerups & ( 1 << PW_INVIS ) ) {
        gun->customShader = cgs.media.invisShader;
        trap_R_AddRefEntityToScene( gun );
    } else {
        trap_R_AddRefEntityToScene( gun );

        if ( powerups & ( 1 << PW_BATTLESUIT ) ) {
            gun->customShader = cgs.media.battleWeaponShader;
            trap_R_AddRefEntityToScene( gun );
        }
        if ( powerups & ( 1 << PW_QUAD ) ) {
            gun->customShader = cgs.media.quadWeaponShader;
            trap_R_AddRefEntityToScene( gun );
        }
    }
}
```

Having a bit of *dèjà vu* yet? If this function looks similar to the previous one, that's because the handling of the powerup visual effect on the gun model is basically the same as the application to the player model. Another `refEntity_t` is passed in, along with a `powerups` variable,

which is tested for the powerup in question. If it exists, the same old rules apply: Assign the shader to the entity's `customShader` member, and pass the entity to `trap_R_AddRefEntityToScene`. I'll bet you can make the appropriate modification to this function with your eyes closed, but just to play it safe, here's the code you'll add:

```

        gun->customShader = cgs.media.quadWeaponShader;
        trap_R_AddRefEntityToScene( gun );
    }
    // if the player holding the gun is draining armor,
    // draw the shell effect on the gun
    if ( powerups & ( 1 << PW_ARMORDRAIN ) ) {
        gun->customShader = cgs.media.armorDrainShader;
        trap_R_AddRefEntityToScene( gun );
    }

```

That should do it! Now you can be assured that either the player receiving the armor drain powerup or another player nearby will have the appropriate visual effect applied to him.

## Regenerating Armor

Now that you have all the code in place to handle the visual effects, you need to come up with a way of actually calling the event and generating armor on the player. Let's jump back to the game code and pop open `ChainLighting_Fire`. Get back into `g_weapon.c` in the game project, and scroll down to line 666, where the damage from the chain lightning is dished out via `G_Damage`. Here's where you will add the powerup flag to the player, and award him with some additional armor:

```

    // Deal out the damage
    G_Damage( target, ent, ent, dir, targettr.endpos, damage, 0,
MOD_LIGHTNING );

    // activate the shell for moment
    ent->client->ps.powerups[PW_ARMORDRAIN] = level.time + 100;

    // increment armor!
    ent->client->ps.stats[STAT_ARMOR]+= 5; // same as a shard

```

Right after `G_Damage` is called, the `playerState_t` variable `ps` has its member `powerups` (which is an array) updated. The index of the array equals the new powerup flag you created, `PW_ARMORDRAIN`, and its value equals the current game time plus 0.1 seconds. That should adequately reflect the shell effect as the player succeeds in chaining targets together. Then, `ps` has another member updated, `stats`, which is also an array. The index in question this time equals `STAT_ARMOR`, which refers to the player's current armor level. Tacking on an additional 5 armor units per chain event is the equivalent of picking up a shard of armor off the map; that's a nice, fair number. Anything less isn't very noticeable, and anything more can accrue armor at an astronomical speed. Note, however, the armor will max out at 50 using this technique.

Let's review. You have the code in `cgame` that will initialize the shader and load it into memory, and assign it a variable to be used as a reference later. Then, you

have added logic to apply the effect to other player models and to the main player's gun model when the powerup is active. Finally, you have added the logic to add the powerup to the player when a chain-lightning event triggers, also adding armor to the player. The final step is physically creating the shader script and image that will be used in the visual effect.

The script file, which needs to exist in `/scripts/` in your `MyMod` folder, contains a single entry:

```
powerups/armordrain
{
    deformVertexes wave 100 sin 3 0 0 0
}
```

## NOTE

### By setting

`ps.powerups[PW_ARMORDRAIN]` equal to the current game time (`level.time`) plus an additional 100 milliseconds, you're effectively telling Q3 how long the effect will last. If you use anything over one second (1000 milliseconds), the visual effect will behave like all other powerups in Q3—that is, as its time runs out, a warning sound alerts the player that the powerup will soon disengage.

```
map textures/effects/armordrain.tga
blendfunc GL_ONE GL_ONE
tcGen environment
tcMod turb 0 0.2 0 1
}
}
```

This shader looks almost exactly like the effect for the quad-damage powerup, except that this has a surface parameter of `turb`, or *turbulence*, causing a swaying and swirling effect on the texture image being used. Then, in the `/textures/effects/` folder, which also needs to exist in MyMod, you will place the image that is to be loaded by `Q3` and used in the shader effect. Both these files are on the CD-ROM in `/files/source/chapter06/chp06-02.zip`, in the PK3 archive file included. You can build your DLLs now, copy them to your MyMod directory (if necessary), and drop the `armordrain.pk3` file in the same folder alongside the DLLs. Remember, you don't have to create the `/scripts/` or `/textures/effects/` directory, because the paths exist within the PK3 file itself. (I'll discuss paths inside the PK3 file in Chapter 8, "Where to Go Next.")

With all your DLLs and the PK3 file in place, load up the new mod and give it a run with some bots. Don't forget to run `Q3` with the `sv_pure` setting set to 0, so that `Q3` will look for DLLs outside of the PK3 file. In Figure 6.4, a bot manages to hit the player and another bot, causing your new armor regeneration shell effect to render.

You're quickly beginning to harness the full effect of the `cgame` code by creating a visual effect that can be assigned to a game-related powerup. Let's push this modification one step farther by accessing yet more frequently used areas of `cgame`: the HUD (heads-up display) and sounds.

## Adding New Icons and Sound Effects

In order to round out the modification of the chain-lightning gun, let's add one more feature to the `cgame` code. One of the items you haven't yet dealt with is the reward notification used when handing out awards to players. These awards show up on the player's screen (like the scoreboard), and do not affect the server-side game code



**Figure 6.4** A Q3 bot chains targets and regenerates armor

when they are visible. Likewise, if an opponent of the player scores the same award, the icon that represents the award hovers above the opponent’s head. Additionally, a sound-effect may be tied to that particular award. So, let’s take a look at displaying a new reward on the HUD (if the player himself gains the award), adding the icon above a player’s head (if an opponent gains the award), and playing a sound effect in either case. The new reward will be given to a player when he manages to make a kill by hitting three players in a chain at once.

## Making It Count

The first part that needs tackling is the server-side portion of the code—the stuff that sits in game. Start by opening up `g_local.h` and scrolling down to line 305, near the end of the definition of `gclient_s`. Here, let’s add a new integer to hold the total number of targets being hit by chain lightning:

```
char      *areabits;
int       currentChainCount;    // for chainlightning;
};
```

Now you will be able to access the `currentChainCount` member via the entity variable that points to the player firing the chain-lightning gun.

Next, the award will require the creation of new variables that will hold references to the award you will add. There are two important places that will need to be updated to hold these new values. You will find the first change in `bg_public.h`, where all the definitions for `eFlags` exist. `eFlags`, as you recall, is a member of the `entityState_t` struct. One or more bit flags can be assigned to an entity's `eFlags`. Remember that because the values are bit flags they can be mixed and matched however you see fit, but a combination of flags must never add up to be *exactly the same value as any other bit flag*.

Scroll up to the top of `bg_public.h` and take a look line 222, where the series of `#define` keywords begin. As you move down the list, you see that a bunch of the variable names near the end refer to awards, such as `EF_AWARD_IMPRESSIVE` and `EF_AWARD_DEFEND`. Notice the numbering scheme that was chosen for the previous defines. They start with a hexadecimal value of `0x00000001`, followed by `0x00000002`, `0x00000004`, and `0x00000008`. After that, the bit flags carry on from `0x00000010`, in the same fashion. The last one in the list, `EF_TEAMVOTED` is `0x00080000`. You could go ahead and add the next one in the list, but due to a limitation in the *Q3* source, let's swap with an existing award. `EF_AWARD_DEFEND` is an award used in *Capture the Flag*. Also, because you are modifying a vanilla deathmatch game, you can swap places with it and your new flag. Make the change so they look like this:

```
#define EF_AWARD_IMPRESSIVE 0x00008000 // draw an
impressive sprite
#define EF_AWARD_TRIPLE 0x00010000 // our triple
award
#define EF_AWARD_ASSIST 0x00020000 // draw a assist
sprite
#define EF_AWARD_DENIED 0x00040000 // denied
#define EF_TEAMVOTED 0x00080000 // already cast a
team vote
#define EF_AWARD_DEFEND 0x00100000 // draw a defend
sprite
```

As you can see, the hex value of `0x00010000` is now being used by `EF_AWARD_TRIPLE`, because you have bumped `EF_AWARD_DEFEND` down to the bottom, giving it a value of `0x00100000`.

Now that you have an award flag that can be applied to the player when he earns the award, you now need a way to count up how many times that particular award has been given to a player. This is used in scoring (especially for tournament play) as well as a visual indicator to the player of how many times he has been given a specific reward (see Figure 6.5).

To keep a running log of how many times this new award is given to a particular player, head to line 214 in `bg_public.h`, where you will see an enum called `persEnum_t`. Because `persEnum_t` is of type enum, you know from experience that the values are all integers, and start at zero. The use of this enum is to help in the description of an array index. The array, `persistant`, is a member of the `playerState_t` struct, which has a special behavior: Unlike many of the other members of `playerState_t`, which are reset after a player dies, the values in the `persistant` array are not cleared. This makes perfect sense, because you do not want a player to have his awards cleared when he dies.

Each index of the `persistant` array will ultimately map to a value, which will indicate how many awards that player has received. For



Figure 6.5 A player accrues a hefty number of Excellent awards



### Bits and Bytes

---

Programmers have a variety of ways to describe numbers in code. The most common ways are decimal, hexadecimal and binary formats. Decimal is the most familiar of the three; it is the base-10 standard that you learned in school, where numbers are represented by combinations of the characters 0 through 9. The next format, used widely across multiple programming languages, is the hexadecimal system, which is base-16. This means that there are an additional six characters used to represent a number. The extra six characters are the letters A through F. The third most used way to describe numbers are closer to a computer's natural language of ones and zeroes, which is called binary. The binary system is base 1, which means there are only two digits needed to represent any binary number: 0 and 1.

To give you a visual representation, here are values 0 through 20, as they are displayed in their native formats:

<b>Decimal</b>	<b>Hexadecimal</b>	<b>Binary</b>
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011

Decimal	Hexadecimal	Binary
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000
17	11	10001
18	12	10010
19	13	10011
20	14	10100

In the C language, programmers have specific ways of working with these various formats. Decimal numbers are handled normally; if you want to deal with the value for 2014, you literally use “2014” in your code. For a hexadecimal value, the number is represented by “0x” symbol, followed by eight digits. To deal with the hex value of 2014, you would code it as 0x000007DE. When you work with bit flags in the Q3 source, you will deal with numbers at the hex level.

As discussed in earlier chapters, the value of a bit flag must not match the sum of any other flags. The common numbering for this scheme is 1, 2, 4, 8, 16, 32, and so on. So, by looking at the hex values of the bit flags listed in this section, you can see that they match the numbering convention perfectly. The first four values are 1, 2, 4, and 8, then 0x00000010, which is 16, 0x00000020, which is 32 . . . I think you can see the pattern emerging.

Because there are eight bits in a byte, and the eFlags variable is a 32-bit integer, you can deduce that the hexadecimal representation of the preceding bit flags indicate one byte for every digit (8 bits in a byte \* 8 bytes = 32-bit integer). This means that 1 is a byte, and so is F (and everything in between).

example, if `persistant[PERS_EXCELLENT_COUNT]` equals 4, then you know that player has accrued four Excellent awards. So, let's go ahead and add a new index to this enum. After the declaration for the `PERS_EXCELLENT_COUNT` variable, add a new variable, like so:

```
PERS_IMPRESSIVE_COUNT,           // two railgun hits in a row
PERS_EXCELLENT_COUNT,           // two successive kills in a
short amount of time
PERS_TRIPLE_COUNT,              // new for triple reward
count!!
```

Now, you have a new index that you can refer to in the `persistant` array, called `PERS_TRIPLE_COUNT`. This index counts how many times the player has been given the new award.

Now that you have the two variables necessary to track the award (the count of the award, and the flag that determines whether it is being applied to the player), you need to put them in place. Open `g_combat.c`, and scroll to line 521. This is right smack in the middle of the `player_die` function, which is called whenever a player perishes in the game. As you may note, there are some awards already being handed out to the player in this chunk of code, such as the Impressive award, given when a player kills his opponent with the gauntlet.

After the code to handle the Excellent award, add the following code so that the function reads as follows:

```
attacker->client->ps.eFlags |= EF_AWARD_EXCELLENT;
attacker->client->rewardTime = level.time + REWARD_SPRITE_TIME;
}

// if our chain lightning caught 3 people add the award!
if ( meansOfDeath == MOD_LIGHTNING && attacker->client-
>currentChainCount == 3) {
```

## NOTE

As the C comment notes on line 200 of `bg_public.h`, you may only ever have 16 values in the `persEnum_t` enum. Anything more will extend beyond the boundaries of the `persistant[]` array, causing mayhem and destruction! (Well, not quite . . . but it will cause some grief.)

```

// play triple award sound
attacker->client->ps.persistent[PERS_TRIPLE_COUNT]++;

// add sprite over player's head
attacker->client->ps.eFlags &= ~(EF_AWARD_IMPRESSIVE |
EF_AWARD_EXCELLENT | EF_AWARD_GAUNTLET | EF_AWARD_ASSIST |
EF_AWARD_DEFEND | EF_AWARD_CAP | EF_AWARD_TRIPLE );
attacker->client->ps.eFlags |= EF_AWARD_TRIPLE;
attacker->client->rewardTime = level.time +
REWARD_SPRITE_TIME;
}

```

Here, a simple check is made to see if the target was killed by MOD\_LIGHTNING, which is the means-of-death flag assigned to the lightning gun's energy beams. Every way a player dies in Q3 has an appropriate means-of-death flag, all of which are declared near line 552 of bg\_public.h. Table 6.2 presents a no-nonsense listing of the some morbid ways in which a Q3 player can meet his maker.

**Table 6.2 Means-Of-Death Flags**

Variable	Value
MOD_SHOTGUN	Death by shotgun blast
MOD_GRENADE	Death by direct contact with a grenade
MOD_GRENADE_SPLASH	Death by a grenade's explosion
MOD_ROCKET	Death by rocket launcher
MOD_PLASMA	Death by plasma gun
MOD_LIGHTNING	Death by lightning bolt
MOD_WATER	Death by drowning
MOD_LAVA	Death by roasting to a crisp in liquid magma
MOD_CRUSH	Death by being crushed (by some obstacle)
MOD_TELEFRAG	Death by another player teleporting <i>into you</i>
MOD_FALLING	Death by falling a great distance
MOD_SUICIDE	Death by killing yourself

After the check for `MOD_LIGHTNING`, another check is made to see if your new member variable, `currentChainCount`, equals 3. If both evaluations turn out to be true, the persistent array is incremented by one on the index equal to your new enum value, `PERS_TRIPLE_COUNT`. Then, the next three lines of code handle placing the new award above the player's head so that other players in the game can see what award was just handed out. The `rewardTime` member of `gclient_s` (the client variable) simply holds the time in which the reward is to be displayed. By setting it to `level.time` (current game time), plus the value of `REWARD_SPRITE_TIME`, `Q3` knows to play the award notification for two seconds (because `REWARD_SPRITE_TIME` is defined as 2000).

Next, you want to make a few adjustments to `ChainLightning_Fire` to use your new target-counting variable. Scroll up to line 594 in `g_weapon.c`, and remove the declaration of `currentTargetTotal`. This variable is no longer needed; you will be using the client's counter instead. Line 603 gets the next change, as so:

```
damage = 8 * s_quadFactor;
ent->client->currentChainCount = 1; //changed lined
chain_targets[0] = target;
```

Now you're setting the `currentChainCount` variable that resides in the client struct. Change line 657 next, so that it reads like this:

```
if (!target->takedamage)
    break;
```

```
chain_targets[ent->client->currentChainCount++] = target; //
changed line
```

Again, you make a reference to the new client counting variable. The rest of the function can remain as is.

One last thing needs to be added to the game portion of the award code. Remember working with `ClientThink_real` a few chapters back? I explained then that `ClientThink_real` was the player's think function, meaning it executed once or twice about every frame of game time (exactly once for each frame on the client). You played with gravity as well as speed, both of which represent values that need to be checked every frame in order to properly place the player in the game. Another check that needs to take place every frame is whether the amount of time during which an award is to be displayed has elapsed;

if so, then the award needs to be removed. This happens on line 807 in `g_active.c`. Head there now, and note that all the existing awards are being turned off in this line of code:

```
client->ps.eFlags &= ~(EF_AWARD_IMPRESSIVE | EF_AWARD_EXCELLENT |
EF_AWARD_GAUNTLET | EF_AWARD_ASSIST | EF_AWARD_DEFEND | EF_AWARD_CAP);
```

All you need to do to keep the player clean and clear after the new award is given to him is to clear the award away when its time is up. Do that by modifying the preceding line so it reads like this:

```
client->ps.eFlags &= ~(EF_AWARD_IMPRESSIVE | EF_AWARD_EXCELLENT |
EF_AWARD_GAUNTLET | EF_AWARD_ASSIST | EF_AWARD_DEFEND | EF_AWARD_CAP |
EF_AWARD_TRIPLE );
```

Super simple! All you do is add a pipe symbol (`|`) and the new flag that represents the triple award you created. Now, each pass through the client's `think` function will check to make sure the award hasn't been hanging around too long.

## Getting Up in the Player's Face

Now it's time to get freaky with the client code. When an award is given to a player, it is splashed up on the player's screen, right on top of the game in action. In a sense, it acts much like the scoreboard, overlaying the action on the HUD. Awards often have sound effects associated with them as well; generally, the deep, foreboding voice of the *Q3* announcer booms out of the speakers when an award is given. This is where you can get creative and make your own sounds to import into the game. Because I am providing you an icon for the new award, I'll also provide you with a sound file used to announce which award is given, but feel free to go off and experiment with your own sounds. Near the end of the book, I'll recommend some great tools to help you in your sound-processing endeavors.

To begin the client modification to handle a new award, you will need to create a handle to the icon and the sound effect. *Handles* are simple integers that have been retyped by the programmers at ID, so that their role is more easily identified. In traditional C and C++ programs, handles simply act as tools that programmers use to quickly identify other processes in their application. Because icons and sound effects are used frequently in *Q3* (and there are a LOT of them), a handle is

an excellent way of working with them. Don't be confused by the terminology: the guts of the handle in *Q3* are simply an integer, and nothing more—and if you don't believe me, look at line 307 in `q_shared.h`!

Start by opening `cg_local.h` and scrolling to line 801. Here, a chunk of handles is being declared in the struct `cgMedia_t`, which becomes a member named `media`, in the global variable `cgs`. This is exactly the same place you created a handle to your `armorDrainShader` a few sections ago. These particular handles will hold references to awards. Go ahead and add a new one after the `medalGauntlet` handle, like so:

### NOTE

`cgs` stands for *client game static*, which represents all data loaded in by *Q3* or calculated by the server updates that are sent during game play.

```
qhandle_t    medalExcellent;
qhandle_t    medalGauntlet;
qhandle_t    medalTriple; // our triple medal!
```

Here, you simply use the ID-created keyword `qhandle_t`, and declare `medalTriple`. Next, scroll down to line 873 and take a peek at the handles being declared for sound effects. Like the new keyword `qhandle_t`, another data type called `sfxHandle_t` is used to declare these sound effects. Add a new one after the sound effect handle for the Excellent award:

```
sfxHandle_t  impressiveSound;
sfxHandle_t  excellentSound;
sfxHandle_t  tripleSound; // our triple sound effect!
```

This stuff is easy! There it is, a variable called `tripleSound`, which is of data type `sfxHandle_t`.

Now that you have handles to your award icon and sound effect, you'll need to have those files cached before *Q3* starts a new game, exactly as you did with your shader earlier on. To cache your new award icon and sound effect, open the file `cg_main.c` and scroll to line 996. Here you can see a familiar struct being assigned values, `cgs.media`. Right after `medalExcellent` in the `cgs.media` struct is assigned a value, go ahead and add an assignment to your `medalTriple` handle:

```

cgs.media.medalImpressive = trap_R_RegisterShaderNoMip(
"medal_impressive" );
cgs.media.medalExcellent = trap_R_RegisterShaderNoMip(
"medal_excellent" );
cgs.media.medalTriple = trap_R_RegisterShaderNoMip( "medal_triple"
); // our new triple medal!

```

Here, you access your `medalTriple` handle from the `cgs.media` struct, and assign it the value returned from the function

`trap_R_RegisterShaderNoMip`, passing in a string equal to the name of the shader. This function is a system call, which acts very similarly to the `trap_R_RegisterShader` function used earlier, except that this particular shader tells *Q3* that the image involved will not be mip-mapped.

After preparing your award icon to be cached, go ahead and do the same for your sound effect. Scroll up to line 641 in the same file, and you should see a bunch of sound-effect handles getting similar treatment. Add a new assignment to your `tripleSound` handle, right after `excellentSound`'s assignment:

```

cgs.media.impressiveSound = trap_S_RegisterSound(
"sound/feedback/impressive.wav", qtrue );
cgs.media.excellentSound = trap_S_RegisterSound( "sound/feed-
back/excellent.wav", qtrue );
cgs.media.tripleSound = trap_S_RegisterSound(
"sound/feedback/triple.wav", qtrue ); // our triple sound effect

```

In this bit of code, you assign `tripleSound` (which is also a member of `cgs.media`) the return value of yet another system call,

## TIP

*Mip-mapping* refers to the technique of applying different qualities of a texture to a given 3D surface, based on the distance of that 3D surface from the viewer. Typically, in a mip-mapped scenario, a 3D image that is far away from the viewer has a small, less-detailed texture applied to it, whereas a surface that is close to the viewer gains a much larger and more precise texture. This often conserves CPU cycles that would otherwise be needed to calculate what the texture would look like at any given distance. *Mip* itself is an acronym, *multum in parvum*, which is Latin for "many in one."



`trap_S_RegisterSound`, which will tell the *Q3* engine to set aside some memory for a sound clip. The function `trap_S_RegisterSound` takes two parameters: a string equal to the relative path and file name that physically holds the sound effect to be played, and a `qboolean`, which indicates compression for the file.

## Cache Money

As the heading suggests, this is the time for your cache to “pay off” (sorry, bad pun). You now need to implement the code necessary to pull the icon and sound effect from the cache and present them to the player when he receives the award. The icon that represents the award must be placed on the HUD and also be made to hover above the player’s head, so that other players can witness the event from afar. In addition, the sound effect indicating that the award was given will have to be played.

Start by opening `cg_playerstate.c` and scrolling to line 354. At this point in the file, you will be in the middle of `CG_CheckLocalSounds`, a giant function used on the client to play sound effects. Many sound effects can simply be played without being attached to a certain visual effect, but in this case, you will play a sound and display an icon. Luckily for you, this functionality is wrapped up nicely in a function called `pushReward`:

```
static void pushReward(sfxHandle_t sfx, qhandle_t shader, int
rewardCount)
```

As you can see, `pushReward` takes three input parameters. The first is an `sfxHandle_t`, the second is a `qhandle_t`, and the third is an integer value representing the number of awards to be dished out, held in a variable called `rewardCount`. On line 354, right after the Excellent award is handled, add the following bit of code:

```
pushReward(sfx, cgs.media.medalExcellent, ps->persistant[PERS_EXCEL-
LENT_COUNT]);
    reward = qtrue;
    //Com_Printf("excellent\n");
}

// play the sound!
if (ps->persistant[PERS_TRIPLE_COUNT] != ops-
```

```

>persistant[PERS_TRIPLE_COUNT]) {
    sfx = cgs.media.tripleSound;
    pushReward(sfx, cgs.media.medalTriple, ps-
>persistant[PERS_TRIPLE_COUNT]);
    reward = qtrue;
}

```

In the same manner as the `PERS_EXCELLENT_COUNT` index of `persistant` is handled, you will check to see whether `ps->persistant` is not equal to `ops->persistant` (as noted by the `!=` symbol). “Okay,” you may say, “but what are `ps` and `ops`, and what do they have to do with each other?” Very good question! In every cycle through a client frame of game time, the current value of the player’s state is held in `ps`, while the previous frame of the player is held in `ops`. So, if an award were to be assigned to the player, you know from handling the preceding game code that the value of the `PERS_TRIPLE_COUNT` index of `persistant` would be one value greater than it was a frame ago, when the award had not been given. This is indicative of the preceding code snippet. In the split second of game time between the player’s last frame and the player’s current frame, an award was given, causing the value of the current player’s award variable to be different from that of the variable one frame ago. I know it sounds complicated, but you have to remember that you are dealing with time at the atomic level when you program logic like this; each tidbit of code is looking at a very small slice of game time, so you can imagine that this function is called many times per second. Comparing the present with one frame in the past is the fastest way of looking at changes made to the player.

Now, back to the function. As noted, if a valid award change has been detected, the local `sfx` variable is assigned to `cgs.media.tripleSound`, which is then passed into `pushReward`, along with `cgs.media.medalTriple`, which is the handle for the award icon. Then, another local variable, `reward`, is set to `qtrue`. `reward` is simply an override flag that tells `CG_CheckLocalSounds` that this particular sound effect is more important than others, and should take precedence over some less-important audible notifications.

The last change you will make is to `cg_players.c`, in the function `CG_PlayerSprites`. This function is responsible for drawing an icon above a player’s head in the *Q3* world, so that it hovers in place, indicating the award that the particular player just received. Scroll down

to line 1932, after the handling of the Excellent award, and add the following code:

```
if ( cent->currentState.eFlags & EF_AWARD_EXCELLENT ) {
    CG_PlayerFloatSprite( cent, cgs.media.medalExcellent );
    return;
}

// float the sprite!
if ( cent->currentState.eFlags & EF_AWARD_TRIPLE ) {
    CG_PlayerFloatSprite( cent, cgs.media.medalTriple );
    return;
}
```

As you can see, immediately after the Excellent award is handled, a check to the `cent->currentState.eFlags` variable is checked. It just so happens that, like the visual shell effect earlier, `cent->currentState` will map to the state of a specific entity in the game code. Because you applied the `EF_AWARD_TRIPLE` to the attacker entity in `player_die`, you can now look for that flag by accessing `cent->currentState`. If the proper flag is detected, a call to `CG_PlayerFloatSprite` is made, passing in the entity (the attacker) and the icon to hover above the player's head, held in the handle `cgs.media.medalTriple`. `CG_PlayerFloatSprite` is a function that wraps up the technical details for you, determining the physical location of the vector above the player's head, assigning the proper shader, and so on; you can find its definition on line 1876 of `cg_players.c`

That's it! All you need to do now is compile, and place the appropriate icon and sound effect into their respective folders in your MyMod directory. If you are using the PK3 file I supplied on the CD, you can place it in the MyMod folder on its own. If, however, you want to use your own icon and sound effect, you will need to place an image called `medal_triple.tga` in `/MyMod/menu/medals/`, and a sound file called `triple.wav` in `/MyMod/sound/feedback/`. Additionally, because the icon is a graphical image represented in the world of *Q3*, it needs a shader associated to it. I provided a shader script in the PK3 file called `triple.shader`, but feel free to make your own. Place it in `/MyMod/scripts/`, with the contents of the script reading as follows:

```
medal_triple
{
```

```
nopicmip
{
    clampmap menu/medals/medal_triple.tga
    blendFunc blend
}
}
```

This is a very simple shader, telling *Q3* that the image is not to be mip-mapped, and is not to be repeated if the dimensions of the image grow or shrink. The image will be resized, of course, because it will be hovering over a player that could be close in the foreground or very far away. `blendFunc blend` simply indicates that this image has transparency. With these files in place and your DLLs all compiled, give your new mod a try. Throw in some bots and see what happens when you kill them with the chain-lightning gun. Figure 6.6 shows a player who has already accrued two of the triple-kill awards:

As you can see, adding an award is a pretty simple feat, as long as you understand that the files you use must be cached ahead of time. You can now go on to play with other similar types of icons and sound



Figure 6.6 A player receiving the new triple-kill award

additions to your mods, simply by playing around with some of the techniques you've learned in this section. Try experimenting by changing the rules that dictate how an award is handed out.

## Summary

There was a lot to get through in this chapter, but by now, you should be up to speed on how important the `cgame` code is to making mods in *Q3*. `cgame` is integral to the entire process, and should not be overlooked when developing code that seems to be needed only on the server side. As you learned in this chapter, the `cgame` code is responsible for creating visual effects in the form of events, which can tie directly to states of entities as they appear on the server, such as a gun firing a lightning bolt. You also saw that `cgame` code could be used to manipulate shaders and create icons on the screen (and in the game), as well as play sound effects. All this information will be important to you as you move to the third section of this book, which binds together all you have learned up to this point. For now, let's move to the next chapter, which will teach you how to work with *Q3*'s user interface.



# CHAPTER 7

# DEFEND THE FLAG

This chapter focuses on the development of your new mod, *Defend the Flag*. This consists of implementing a new game type and system of rules into *Q3*, based on the existing game type of *Capture the Flag*. You will create a set of unique rules, which will dictate how the game is played. You will also learn about level entities and how they can be changed to reflect new types. This will give you the ability to modify existing *CTF* flags—transforming them into *sigils*, the new item that will be used in *DTF*—utilizing a few clever tricks in the process.

## The Rules

*Defend the Flag* is an exciting team-based mod that alters the game play slightly from standard *CTF* style. In *DTF*, a team of two will battle for control of three flags, placed in various spots throughout a map. In standard *CTF*, the goal of the player is to race to the opponent's flag, steal it, and return it to his own base. In *DTF*, however, there is no particular team flag (in other words, no red-team flag, and no blue-team flag, either). Instead, the flags are neutral, belonging to neither team. In addition, instead of two flags, there will be three.

As the match begins, players from either team race to the three neutral flags and attempt to tag them. When a player successfully touches a flag, it is converted to the color of the team that tagged it. The goal, then, is to have all three flags held by a given team. Every few seconds, each held flag earns its team a number of points; the more flags held, the more points accrued at each interval. If an opposing team member manages to touch the held flag, it converts to the color of the new team, at which point the new team will be the recipient of accrued points.

The *DTF* mod is a popular game type, and has found its way into other mods, going by other names such as *Capture-and-Hold*, or *Domination*. *Urban Terror* is a version of this game type, implementing

more than three flags per level. Some professionally developed games have also used variations on this game type, such as *Tribes* and *Unreal Tournament*. In Figure 7.1, you can see a game of *Domination* being played in *Unreal Tournament*, another popular FPS game. Instead of flags, *Domination* uses control points—floating symbols that change shape and color when they are tagged by a team.

This is a perfect game-type mod to implement because many of the existing bits of logic for *CTF* can be reused. This will ultimately save you hours of development time, because you won't have to re-write the game's specifics from scratch. From the preceding summary, let's extract the pertinent information and create a list of goals for your mod:

- Place three flags in a map instead of two.
- Modify flag behavior so they are not picked up when touched.
- Develop a way to accrue points, at specific intervals, based on the number of held flags.
- Update the HUD to show which flags are held, and by which team.



**Figure 7.1** Players guard a control point in *Domination*



- Update the HUD to give the player three “compass pointers” showing the way to the flags (and give the user an option to turn it off).
- Update the user interface to allow players to select *DTF* as a game type and choose existing *CTF* maps to play on.

Not a bad set of goals, don't you think? Now that you have your list of goals for the mod, let's take a look at what it will take to implement them.

## What You Will Reuse

For starters, you should probably be aware that flag models already exist in *Q3* right off the shelf, because *CTF* is a built-in game type. So breathe easy—you won't have to worry about creating any complicated models or frames of animation. The current *Q3* flag model will work fine for this mod. There are, however, only red and blue skins for these models in the current PK3 files for *Q3*. So you'll definitely need to come up with a third neutral skin.

Secondly, the standard *CTF* maps have spawn points for only two flags in their current state. Because you do not have the luxury of having the source files to the maps used in *Q3*, you cannot simply add a new entity type to the map, recompile, and be good to go. You'll have to come up with another way of generating the third flag point. You'll be using an interesting trick to generate this third flag point. Also, *CTF* maps that were built specifically for the Team Arena Expansion Pack have three *CTF* spawn points (because they must contain a spawn point for one white flag, in the *One-Flag CTF* game type), so you'll make the mod smart enough to use either map type.

Thirdly, as you may know from experience, when you touch a flag in a regular game of *CTF*, the flag disappears from its holder in the map and appears on the player in the form of a powerup. The player then carries this flag around with him. If he is killed, the flag is dropped. If it remains dropped for too long, or it is touched by a teammate, it is sent back to its holder. Unfortunately, this will not do for your mod. For the *DTF* mod, you do not want the player to ever pick the flag up. Furthermore, when a player touches a flag in *DTF*, you will want it to change color. This flag behavior will have to be modified.

As for the remaining specifics, the standard rules of *CTF* will still work fine, as will the variables that hold various *CTF*-related info, such as `capturelimit` (how many captures will be needed to end the game). You'll simply tweak the variables so that they more accurately reflect what a *DTF* game will require. The user interface will also be tweaked to allow for the new *DTF* game type to be selectable, and when the user chooses it, the updated UI will properly constrain the map selection list to existing *CTF* maps.

## What You Will Create

Now, let's discuss what you'll need to generate by hand for this mod. For starters, there is no function that accrues *CTF* points as a game progresses, so that function will have to be written. There are, however, similar functions, and you will take a look at them to see what you can use in the new function.

There is definitely no way to show three flags on the HUD. You'll have to come up with a new layout for the HUD's icons that represent flags in *DTF*. In standard *CTF*, flags are represented on the HUD by icons that indicate whether the flag is at home, stolen by an enemy teammate, or dropped somewhere in the map. For *DTF*, the states of the flags will be different. Now, you'll want to show the three flags, and which team currently holds each. This will be indicated by the color of the icons: red, blue, or your neutral color.

In order to give the player some guidance as to where the flags are on the map, you will need to create compass pointers that hover over the player's view, pointing in the general direction of the flags. This functionality doesn't exist at all in *Q3*; it will be a brand-new chunk of code. You'll implement this new compass function as a client-specific event, because its state does not need to be communicated to the server. As well, you will add appropriate Cvars to `cgame`, to allow the user to disable the pointers if he chooses to do so.

## Preparing *Q3* for *DTF*

To lay the foundation for your new game type, a traversal of the existing code base is in order. Littered throughout the files that comprise

the `game`, `cgame`, and `ui` projects are many variables and initialization routines that prepare *Q3* to handle the unique rules that apply to each specific game type. For example, when a *CTF* game is launched, *Q3* needs to know what a `capturelimit` variable represents, what the “red flag” and “blue flag” entities do, how to spawn players into the map differently, and so on. In this next section, you will begin to modify those appropriate areas of the code base so that your mod is recognized by *Q3* as well.

## Your Journey Begins at `gametype_t`

The heart of all the different types of games that *Q3* supports lie in an enum called `gametype_t`. A Cvar called `g_gametype` holds the value of the game type being played, and this value is communicated between `cgame` and `game` as well. Much of what goes on in *Q3* that requires a game rule lookup is based on the value found in `g_gametype`. You can find the `gametype_t` enum declared on line 79 of `bg_public.h`, and it looks like this:

```
typedef enum {
    GT_FFA,                // free for all
    GT_TOURNAMENT,        // one on one tournament
    GT_SINGLE_PLAYER,     // single player ffa

    //-- team games go after this --

    GT_TEAM,              // team deathmatch
    GT_CTF,               // capture the flag
    GT_1FCTF,
    GT_OBELISK,
    GT_HARVESTER,
    GT_MAX_GAME_TYPE
} gametype_t;
```

As you can see, `gametype_t` is comprised of a series of integers, identified by the `GT` naming convention. The first game type, `GT_FFA`, stands for a free-for-all deathmatch, in which all players fight each other to the death. `GT_TOURNAMENT`, the second value, is for hosting a one-on-one tournament, in which two players battle each other until one player reaches a specific frag limit (or time runs out). `GT_SINGLE_PLAYER` is the

variable set aside for the single-player version of *Q3*, where one human battles through a series of levels against an assortment of bots.

The next set of `gametype_t`'s members represents the team-based styles of play supported in *Q3*. `GT_TEAM` stands for a team-based version of deathmatch, where two teams battle each other for supremacy. `GT_CTF` is the flag (if you'll pardon the pun) that represents the classic game type of *Capture the Flag*, which I have already discussed in great detail. The next variables represent special game types featured in the Team Arena Mission Pack for *Q3*. `GT_1FCTF` is a unique variation of *Capture the Flag* called *One Flag CTF* in which a third white flag exists somewhere near the center of the map. Players must attempt to grab the white flag and use it to touch their enemy's flag for a score. `GT_OBELISK` represents the *Obelisk* game type, in which two teams have a skull-shaped artifact, or obelisk, at each base. Players must attempt to infiltrate the enemy's base and shoot their opponent's obelisk until it explodes. Small scores are obtained for making shots on the obelisk, and a giant team score is awarded when an obelisk is destroyed.

The next game, `GT_HARVESTER`, is even more exciting than the previous types. In a game of *Harvester*, players kill their opponents, producing skulls near the center of the map (the skulls are ejected from a special container). Players must then rush to collect these skulls and take them to their enemy's base to a drop-off point. There is no limit to the number of skulls a player can carry, so courageous players can take in a string of skulls—if they so choose. If a player carrying skulls is killed, all the carried skulls are lost. *Harvester*, shown in Figure 7.2, is probably the most exciting game type featured in the Team Arena Expansion Pack.

The final value, `GT_MAX_GAME_TYPE`, is simply used as total number of game types, often placed in array declarations as the upper limit of that array. To add a new game type to *Q3*, the first change to be made will be here.

Right between the `GT_CTF` and `GT_1FCTF` values, go ahead and add a new entry for *Defend the Flag*, which I will herein abbreviate as *DTF*. An excerpt of the updated `gametype_t` should read like this:

```
GT_CTF,           // capture the flag
GT_DTF,           // defend the flag
GT_1FCTF,
```

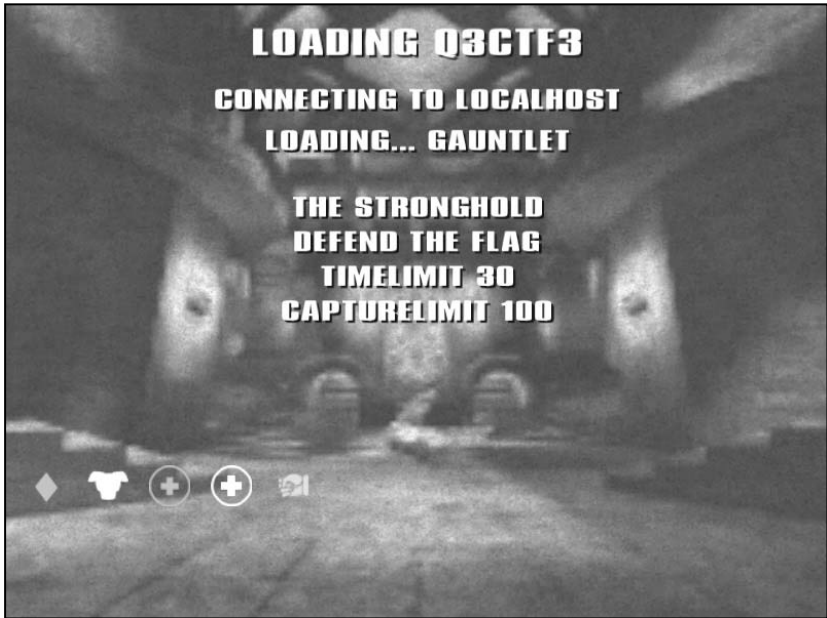


**Figure 7.2** The Harvester game type in the Team Arena Mission Pack

Hooray! The creation has begun! You're going to be seeing a lot more of GT\_DTF, so you'd better make friends with it. In fact, you can use GT\_DTF immediately to describe the game type to new players who try your mod. If you jump to the `cgame` code, open a file called `cg_info.c`, and scroll down to line 233, you will see a set of strings being assigned to a variable `s` based on the game type. This string is then displayed to the player as a level loads, via the function `CG_DrawInformation`. Right after the assignment for `GT_CTF`, add a case for your new `GT_DTF` flag, like so:

```
case GT_CTF:
    s = "Capture The Flag";
    break;
case GT_DTF:
    s = "Defend The Flag";
    break;
```

Now players will know when they are about to start a game using your new mod, as shown in Figure 7.3.



**Figure 7.3** What a player will see during a level load for Defend the Flag

## itemType\_t: Birth of the Sigil

Earlier, I promised that I would show you how to modify the behavior of the existing red and blue flags in *CTF*. In order to prepare for that change, you need to be able to tell *Q3* that the flags are a new type of item. In every *Q3* level, there are items that can be picked up, such as boxes of ammo, weapons, powerups, and (you guessed it) flags. All these items fall into a certain category, which is held in an enum called *itemType\_t*, found near line 596 of *bg\_public.h*. A quick visit to that line number in *bg\_public.h* reveals the following declaration:

```
typedef enum {
    IT_BAD,
    IT_WEAPON,          // EFX: rotate + upscale + minlight
    IT_AMMO,            // EFX: rotate
    IT_ARMOR,           // EFX: rotate + minlight
    IT_HEALTH,          // EFX: static external sphere + rotating internal
    IT_POWERUP,         // instant on, timer based
                       // EFX: rotate + external ring that rotates
    IT_HOLDABLE,       // single use, holdable item
}
```

```

// EFX: rotate + bob
IT_PERSISTANT_POWERUP,
IT_TEAM
} itemType_t;

```

As with all good enums, this declaration maintains a list of variables that define each category of item that exists in *Q3*. You should be able to recognize many of these variables simply from their names:

`IT_WEAPON` refers to weapons, `IT_AMMO` indicates boxes of ammunition, `IT_HEALTH` is a reference to items that give the player health, and so on. One of these variables is incorporated into every item, held in the `giType` member of the `gitem_t` struct.

Because each item in a *Q3* level has this `giType` member indicating the item category it belongs to, you can imagine that there are various bits of logic in the code base that look at `giType`, and perform differently, based on the item being referenced. For instance, when the player touches a health pack he gains health; likewise, when the player picks up ammunition, that type of ammo is added to the player's inventory. Grouping items into `itemType_t` categories make it easier to develop reusable functions for similar items.

### TIP

If you haven't found it yet, the declaration of `gitem_t` can be found on (or near) line 614 of `bg_public.h`. All the items that exist in a level are of type `gitem_t`, and they are poured into a global array called `bg_itemlist`, which you will learn in the next section. If you have a reference to an entity, you can determine whether it is an item by looking at its `item` member, such as `ent->item`.

The flags in *DTF* will, essentially, still be the flags that players have come to know and love. That is, they will look like normal *CTF* flags on the outside. Internally, however, they will act differently, as if they are completely new items (and in fact, they will be). To create this new category of item, insert a new `IT` flag at the end of `itemType_t`, and call it `IT_SIGIL`:

```

IT_PERSISTANT_POWERUP,
IT_TEAM,
IT_SIGIL // DTF Flags will be called 'sigils'
} itemType_t;

```

This new *DTF* flag, which will behave differently from normal *CTF* flags, will be categorized by the variable `IT_SIGIL`.

With `IT_SIGIL` in place, you now have a way of referring to your new *DTF*-specific flags. From now on, I will refer to the flags in *DTF* as *sigils*; when I say “flag,” you’ll know I’m talking about standard *CTF*.

## NOTE

I chose the variable name `IT_SIGIL` because I didn’t want to cause any confusion with existing variables in the game that reference the word “flag” in some way. The new *DTF* flags will be completely separate from the flags used in *CTF*; only their models and skins will be the same. I took the word “sigil” from the names of the symbols used in the original *Quake*, which also had their functionality reprogrammed when they were turned into runes for the first release of *CTF*.

## Fleshing Out the Sigil

In regular *CTF*, flags get pretty active. At one moment, they may be resting idle, safe within the walls of their team’s base. In another moment, they can be stolen by the enemy, strapped to a player’s back as the defending team rushes the thief, laying down a barrage of fire. The flags can also be dropped somewhere in the map—if a carrier is killed during a flag run, for example—waiting patiently to be picked up by another foe or returned to safety by a teammate. In a game of *One-Flag CTF*, the white flag has the added capability of being snatched by either team. Because the flag in *CTF* gets busy at times, *Q3* needs a way to determine what the status of that flag is. That status can be extracted from an enum called `flagStatus_t`, found on line 1374 of `q_shared.h`:

```
typedef enum _flag_status {
    FLAG_ATBASE = 0,
    FLAG_TAKEN,           // CTF
    FLAG_TAKEN_RED,      // One Flag CTF
    FLAG_TAKEN_BLUE,     // One Flag CTF
    FLAG_DROPPED
} flagStatus_t;
```



As shown, the values of a `flagStatus_t` variable can be one of five types. `FLAG_ATBASE` simply means the flag is safe at home. `FLAG_TAKEN` refers to a flag that has been picked up by an enemy; both `FLAG_TAKEN_RED` and `FLAG_TAKEN_BLUE` refer to this same status, except that they apply directly to the white flag in *One-Flag CTF*. The last value, `FLAG_DROPPED`, is the status of a flag that was dropped by a flag carrier.

In *DTF*, the sigil will have three different statuses. It can be white, meaning neither team has touched it; this will be the status of a sigil when a new *DTF* game begins. Additionally, the sigils will change color based on the team that touches them. Because there are two teams in *Q3*, the other two statuses of the sigils will be either red or blue. Under the declaration of `flagStatus_t`, create a new enum that will hold the status of the sigils in *DTF*, calling it `sigilStatus_t`:

```
typedef enum _sigil_status {
    SIGIL_ISWHITE = 0,    // DTF
    SIGIL_ISRED,         // DTF
    SIGIL_ISBLUE        // DTF
} sigilStatus_t;
```

This new `sigilStatus_t` enum will define a variable that will be used to communicate the status of the three sigils between the game and cgame code, functionality that you will be creating later. While I'm on the subject of the three sigils: you will want to indicate to *Q3* that the *DTF* game type will, in fact, have three sigils in each game. Place one more line of code into `q_shared.h` that will identify those three sigils directly above the `sigilStatus_t` declaration you just added:

```
#define MAX_SIGILS    3
```

Now you have a convenient variable that references the total number of sigils that can exist in a game of *DTF*. The variable name is easy to remember; plus, if you later decide that you want more than three sigils in a game of *DTF*, you can change this one line of code. This saves countless hours of weeding through the source trying to change your 3 to some other value, making sure all instances are taken care of. You will get a much more concise result if you search your code base for `MAX_SIGILS` than if you search for the number 3.

## Bending the Rules with powerup\_t

Sometimes rules are meant to be broken. You should be comfortable knowing that in order to get some work done, you may have to cheat a little. The cheating that I am referring to will come in the form of a modification to `powerup_t`. Resting quietly on line 249 of `bg_public.h` is the declaration of `powerup_t`, which reads as follows:

```
// NOTE: may not have more than 16
typedef enum {
    PW_NONE,

    PW_QUAD,
    PW_BATTLESUIT,
    PW_HASTE,
    PW_INVIS,
    PW_REGEN,
    PW_FLIGHT,

    PW_REDFLAG,
    PW_BLUEFLAG,
    PW_NEUTRALFLAG,

    PW_SCOUT,
    PW_GUARD,
    PW_DOUBLER,
    PW_AMMOREGEN,
    PW_INVULNERABILITY,

    PW_NUM_POWERUPS
} powerup_t;
```

As you can see, `powerup_t` is no different from any other enum; it simply has a list of variables that map to integer values, starting at 0 and incrementing by one for each member. You should be able to see variables referencing the existing *CTF* flags, `PW_REDFLAG`, `PW_BLUEFLAG`, and `PW_NEUTRALFLAG`. As well, the last member of the enum refers to the total number of members within it: `PW_NUM_POWERUPS`.

`powerup_t` variables are used very discreetly throughout the *Q3* code base; there are no actual variables that are declared of type `powerup_t`. There is one function, `BG_FindItemForPowerup`, which takes a `powerup_t` variable as an input parameter and returns a `gitem_t`. The reason for this is that because the `powerup_t` data type holds integers, it can be mixed and matched with other enums (such as `weapon_t`) to identify the type of item being referenced (much in the same way that `IT_SIGIL` refers to a category of item, as do other `IT` values).

The most interesting part of `powerup_t` is the C comment at the top of the declaration, indicating that there can be only 16 values in the enum. This stems from the existence of the `powerups` array, held in `playerState_t`. If you open `q_shared.h` and scroll to line 1169, you will see that `powerups` is an integer array that has its upper limit initialized to `MAX_POWERUPS`, which itself is 16. `playerState_t` is one of those touchy variable types that are transmitted between `game` and `cgame` to assist with player movement and prediction. Because `game` and `cgame` can potentially live on two different computers (as discussed in Chapter 5), the data required in each transmission needs to be as small as possible. Hence, smaller physical limits are imposed on the size of the arrays holding info such as stats, ammo, and `powerups`.

Because all 16 values in `powerup_t` are already called for, how can you modify this enum? The answer lies within how the `powerup_t` enum will be used in *DTF*. Because *CTF* flags are actually present on the player when they are stolen, the `cgame` code needs to be able to see the communicated `powerup_t` variable in the player's persistent `playerState_t` member, and update any client-side effects appropriately. In a game of *DTF*, however, the sigils will not act like flags in that they will not be picked up and carried the way standard *CTF* flags are. Because `cgame` will not need to know (or care) about a client-side effect related to the sigils, you can safely make an addition to this enum to represent the sigil.

## NOTE

There will definitely be data in *DTF* that `cgame` will need to see in order to make appropriate updates to the client's view of the game, such as the status of each sigil (white, red, or blue), but you will not be using the `powerup_t` enum for that. Instead, you're going to be using a *config string*, which you'll learn about in the next section.

Go ahead and add the following code near the end of the `powerup_t` enum, so that it reads as follows:

```
PW_INVULNERABILITY,  
  
PW_SIGILWHITE,  
PW_SIGILRED,  
PW_SIGILBLUE,  
  
PW_NUM_POWERUPS
```

```
} powerup_t;
```

There's no turning back now; you're breaking every rule in the book! Just kidding. Bending the rules every once in awhile is a good thing, as long as you're clear on what residual effects your rule-bending will have on the code. This particular modification to `powerup_t` will be safe, as long as you do not try to use the new `PW_SIGILWHITE`, `PW_SIGILRED`, and `PW_SIGILBLUE` variables as indexes to any arrays that have already been sized.

## Transforming Flags into Sigils

In this section, you will learn how to change the behavior of the *CTF* flags so that they become the new *DTF* sigils. Flags, at their core, are items, sharing the same kinds of attributes as those found in weapons, runes, boxes of ammo, and so on. They must have their own models, and sometimes their own skins. They may also have associated sound effects that play when the items are picked up, and they are assigned a unique identifier for both their item type and category. You will use all the preceding information to build the new sigil items in *Q3*, reusing some flag information, thereby creating the illusion that you have altered how *CTF* flags behave. Creating illusions, as you are discovering, is the magic of a mod programmer.

### Every Item Is a `gitem_t`

As explained earlier, the sigil is going to be the heart of *DTF*. In your new mod, players will race around the map attempting to control

three sigils by touching them, which will change the sigil's coloring to match the team that "holds" them. Players will then attempt to defend those held sigils, preventing any enemies from making a similar hold attempt. Because the sigils will have an entirely new behavior, you will create a new item definition for them.

When a map is first launched in *Q3*, all the map's items are parsed and set up to be placed in the world. Each item is compared to a list of items that is declared in `bg_misc.c`—an array called `bg_itemlist`, which is of type `gitem_t`. The initialization of `bg_itemlist` is one giant chunk of code. Because `bg_itemlist` is an array, each element must have default values that map to the members of a `gitem_t` struct. Let's take a look at the definition of a `gitem_t` now:

```
typedef struct gitem_s {
    char      *classname;    // spawning name
    char      *pickup_sound;
    char      *world_model[MAX_ITEM_MODELS];

    char      *icon;
    char      *pickup_name;  // for printing on pickup

    int       quantity;     // for ammo how much, or duration of
powerup
    itemType_t giType;      // IT_* flags

    int       giTag;

    char      *precaches;    // string of all models and images this
item will use
    char      *sounds;      // string of all sounds this item will use
} gitem_t;
```

As shown here, the first member is `classname`, a pointer to a `char`, which will contain the physical name of the item, such as `item_armor_combat` or `weapon_gauntlet`. The second member is `pickup_sound`, another `char` pointer that contains a string, which maps to a sound file. This sound file is then played when the player picks up the item. The third member is `world_model`, which is array that can contain four elements (because `MAX_ITEM_MODELS` equals four). These four elements represent the 3D models used to represent the item in the world.

The next member in `gitem_t` is called `icon`, yet another pointer to a char (see a trend starting here?), and within it is a mapping to an image that will be displayed on the player's HUD when he picks up the item. `pickup_name` holds the display name of the item, as it will be written on the HUD of the player when the item is picked up. The next member is not a char pointer, but instead is an integer called `quantity`, which represents how many units of a particular item are assigned to the player's inventory when one is picked up. This is commonly used for ammunition; when you pick up an ammunition-type item, you may get 50 or 20 of those items instead of one. `quantity` can also refer to an amount of time, such as how long a powerup has until it times out, as is the case with quad damage and the flight powerup.

The `giType` and `giTag` members reference the categories and unique identifiers of the item, respectively. This is where `itemType_t` and `powerup_t` values come into play. The final two members, `precaches` and `sounds`, are used to contain mappings to models, images, and sound files that need to be cached for each item during level loads. Now that you have a feel for what goes into each item, let's take a look at the definition for a *CTF* flag—namely, the red flag:

```
{
    "team_CTF_redflag",
    NULL,
    { "models/flags/r_flag.md3",
      0, 0, 0 },
    /* icon */      "icons/iconf_red1",
    /* pickup */   "Red Flag",
    0,
    IT_TEAM,
    PW_REDFLAG,
    /* precache */ "",
    /* sounds */  ""
},
```

Because all items in `bg_itemlist` are defined during the array's declaration, C programming requirements dictate that each element be identified by curly braces (the `{` and `}` symbols). The preceding code snippet represents one element in the `bg_itemlist` array. Each member of the `gitem_t` struct is defined, in order, to match the declaration of `gitem_t` on line 614 of `bg_public.h`.

As you can see, the classname is `team_CTF_redflag`, there is no default `pickup_sound`, and only one 3D model is assigned to the flag: the model found at `models/flags/r_flag.md3`. An icon is displayed on the player's HUD when a flag is picked up, denoted by the image found at `icons/iconf_red1` (if there is no extension, it is assumed that the image file is a TGA file). Also, the words "Red Flag" will be rendered to the screen when the item is picked up.

You can see that there is a 0 quantity assigned to the flag, because it doesn't technically increment any ammunition or time. The next two values represent the category and type of item, identified by `itemType_t` and `powerup_t` values (`IT_TEAM` and `PW_REDFLAG`). No additional models, images, or sounds are needed for the caching of the flag. Now, when a new map loads, *Q3* can parse the values in this element and handle this item appropriately when it is to be placed in the world.

## Sigils Become New Items

Let's go ahead and make the sigil an official item in *Q3*. Using what you have just learned about `gitem_t` variables and the global array `bg_itemlist`, you should know that in order to create your new items, you will place them into this array declaration in the same format as the previous items. The sigils in *DTF* will reuse the flag models and icons from *Q3* as well as the white flag model and icon from the Team Arena Mission Pack. Scroll down to line 637 in `bg_misc.c`, right after the definition of the `team_CTF_blueflag` item, and add the following lines of code:

```
/* team_DTF_sigil_red
Only in DTF games
*/
{
    "team_DTF_sigil_red",
    NULL,
    { "models/flags/r_flag.md3",
      0, 0, 0 },
/* icon */      "icons/iconf_red1",
/* pickup */   "Flag",
    0,
```

```
        IT_SIGIL,
        PW_SIGILRED,
/* precache */ "",
/* sounds */ ""
    },

/* team_DTF_sigil_blue
Only in DTF games
*/
    {
        "team_DTF_sigil_blue",
        NULL,
        { "models/flags/b_flag.md3",
          0, 0, 0 },
/* icon */      "icons/iconf_blu1",
/* pickup */   "Flag",
        0,
        IT_SIGIL,
        PW_SIGILBLUE,
/* precache */ "",
/* sounds */ ""
    },

/* team_DTF_sigil
Only in DTF games
*/
    {
        "team_DTF_sigil",
        NULL,
        { "models/flags/n_flag.md3",
          0, 0, 0 },
/* icon */      "icons/iconf_neutral1",
/* pickup */   "Flag",
        0,
        IT_SIGIL,
        PW_SIGILWHITE,
/* precache */ "",
/* sounds */ ""
    },
```



This code snippet reveals that the sigils will indeed utilize the existing *CTF* flag models from *Q3* and the Mission Pack. The class name `team_DTF_sigil` will be the actual entity that is held in a map, requiring the item to be spawned (because sigils will be white by default when the game begins). The third element in this array declaration snippet refers to that sigil. You can also see that all three sigils will display the string “Flag” on the player’s HUD when touched; you do this because you want to maintain the illusion that players are dealing with the standard *CTF* flags. Remember, even though these sigils are three unique items in the code, the player will think of them as “one flag that can be colored three ways.” So, making each item announce itself as a flag, without a specific color, is a good way to stop any confusion players may have.

You should also see that each sigil is assigned the `IT_SIGIL` `itemType_t` variable that you set up in the first part of this chapter. This will allow you to control the behavior of all three items in a similar fashion. Finally, your `PW_SIGILRED`, `PW_SIGILBLUE`, and `PW_SIGILWHITE` `powerup_t` variables make their debut as well. They are assigned to each of the three types of sigils that can be represented in the game of *DTF*, and this will assist you later in dealing with logic that must apply to each sigil individually (for example, did a red team member just touch a sigil that was blue?).

## NOTE

A good portion of the functionality you’re building into *DTF* is based on Anthony Jacques’ mod *Domination*, which itself is based on the *Domination* game type found in *Unreal Tournament*. Many of the functions and variables you’ll use in this and the next chapter are modeled after his successful implementation. You can find Jacques’ mod on the Internet at <http://www.planetquake.com/domination/>.

## Initializing Sigils for *DTF*

Your first exploration into using these new items in *Q3* will be to properly initialize and prepare them for a game of *DTF*. Most of the team-related functions are held in `g_team.c` (and its header, `g_team.h`), so you’ll be placing code there for the sigils as well. Your first modification will be within the function `Team_InitGame`, called during the setup of a team-based game in *Q3*. Open `g_team.c`; near the top of the file,

you should see a declaration of the struct `teamgame_t` and a variable `teamgame` that is defined to be of that data type:

```
typedef struct teamgame_s {
    float        last_flag_capture;
    int          last_capture_team;
    flagStatus_t redStatus;      // CTF
    flagStatus_t blueStatus;     // CTF
    flagStatus_t flagStatus;    // One Flag CTF
    int          redTakenTime;
    int          blueTakenTime;
    int          redObeliskAttackedTime;
    int          blueObeliskAttackedTime;
} teamgame_t;
teamgame_t teamgame;
```

Here, the `teamgame_t` struct is comprised of members that reference the status of certain items in various team-based games. You should recognize the `flagStatus_t` struct, because I discussed it earlier in this chapter. The variable `teamgame` is declared to be of type `teamgame_t`, which is then used throughout `g_team.c` to reference the status of the members found in this struct. Tracking the status of your sigils in a game of *DTF* is a prerequisite for this mod, and because *DTF* qualifies as a team game, there is no better place to declare it than here in `teamgame_t`. Go ahead and add a final member, right after `blueObeliskAttackedTime`, which will finish off the struct like this:

```
    int          blueObeliskAttackedTime;
    dtf_sigil_t  sigil[MAX_SIGILS]; // DTF
} teamgame_t;
```

Excellent. You will now have access to `teamgame.sigil[0]`, `teamgame.sigil[1]`, and `teamgame.sigil[2]`, the three sigils needed in a game of *DTF*. However, there's a problem: What in the heck is a `dtf_sigil_t` data type? Because the sigil in *DTF* is going to require a bit more information than simply a status, you'll need to create a struct that contains several members. Go ahead and add the following struct declaration above `teamgame_t`:

```
typedef struct dtf_sigil_s
{
    gentity_t    *entity;
    sigilStatus_t status;
} dtf_sigil_t;
```

Ah, that's much better. Now you can safely use the `dtf_sigil_t` type to declare variables. This simple struct has only two members, a `gentity_t` pointer called `entity`, and a `sigilStatus_t` called `status`. Knowing this, you should now be able to access `teamgame.sigil[x].status`, where `x` is any number from 0 to 2; the result will be one of three values:

`SIGIL_ISWHITE`, `SIGIL_ISRED`, or `SIGIL_ISBLUE`.

Next, let's take a look at `Team_InitGame`, which you should find around line 31. `Team_InitGame` is assigned the task of preparing the `teamgame` variable (and its various members) for a new team-based game, depending on the particular game type that was picked. The first chunk of `Team_InitGame` looks like this:

```
void Team_InitGame( void ) {
    memset(&teamgame, 0, sizeof teamgame);

    switch( g_gametype.integer ) {
    case GT_CTF:
        teamgame.redStatus = teamgame.blueStatus = -1; // Invalid to
force update
        Team_SetFlagStatus( TEAM_RED, FLAG_ATBASE );
        Team_SetFlagStatus( TEAM_BLUE, FLAG_ATBASE );
        break;
```

The call to `memset` should be familiar; it's always a good idea to initialize complex variable types to 0 to wipe away any bad data that may be lurking. Next, a switch block is entered, looking at the value of `g_gametype.integer` (the numeric value of the Cvar `g_gametype` discussed earlier). In the first case, which happens to be a value of `GT_CTF`, `Q3` is told to prepare *CTF*-related information. Therefore, it first accesses the `redStatus` and `blueStatus` members of the `teamgame` variable, and sets them both to -1, telling `Q3` that they must be forced into a valid state as soon as possible. Then, two functions are called (one per flag), both called `Team_SetFlagStatus`, setting the value for each flag to be the initial 0 value of `flagStatus_t`, which is `FLAG_ATBASE`.

To modify `Team_InitGame` to handle your new sigils, you will first need to add a new case to the switch. Right after the break in the `GT_CTF` case, add the following code:

```
    case GT_DTF:
        Init_Sigils();
```

```

teamgame.sigil[0].status = teamgame.sigil[1].status =
teamgame.sigil[2].status = -1; // Invalid to force update
Team_SetSigilStatus( 0, SIGIL_ISWHITE );
Team_SetSigilStatus( 1, SIGIL_ISWHITE );
Team_SetSigilStatus( 2, SIGIL_ISWHITE );
break;

```

With the addition of this code, `Team_InitGame` now knows about the new `GT_DTF gametype_t` variable you declared earlier. When the `GT_DTF` value is found, a call is made to `Init_Sigils` (a function you will write shortly), followed by a similar bit of code borrowed from the `GT_CTF` case. All three sigils must be flagged to have their status forced to an appropriate value when the game initializes. To do that, set the status of each sigil to `-1`. Next, make three calls to `Team_SetSigilStatus`, which will be a new function to handle client-side sigil updating. The function will take two parameters: the index of the sigil in the array and the status of the sigil (white, red, or blue) as found in the `sigilStatus_t` enum.

`Team_SetSigilStatus` is going to be used in Chapter 10 (on the CD-ROM), “Enhancing *DTF*,” when you get into updating the HUD and scoreboard, so for now you will leave it alone. If you try to compile your code at the end of this chapter without properly declaring and defining `Team_SetSigilStatus`, however, your compiler will get upset. So, go ahead and prototype it on line 30, right after the prototype for `Team_SetFlagStatus`, like so:

```

void Team_SetFlagStatus( int team, flagStatus_t status );
void Team_SetSigilStatus( int sigilNum, sigilStatus_t status );

```

As for the body of the function, jump down to where `Team_SetFlagStatus` is defined, near line 188, and add the following code above it:

```

void Team_SetSigilStatus( int sigilNum, sigilStatus_t status ) {}

```

Now your compiler will be happy when you attempt to build a DLL later on. You still have to take care of another function that doesn’t exist yet: `Init_Sigils`. Above the empty `Team_SetSigilStatus` function, add the following definition for `Init_Sigils`:

```

void Init_Sigils( void ) {
    gentity_t    *point = NULL;
    int          sigilNum = 0;
}

```

```

for (point = g_entities; point < &g_entities[level.num_entities] ;
point++)
{
    if (!point->inuse)
        continue;

    if (!strcmp(point->classname, "team_DTF_sigil")) {
        teamgame.sigil[sigilNum].entity = point;
        sigilNum++;
    }

    if ( sigilNum==2 )
        return;
}
}

```

Let's take a look at what's going on here. An initial `gentity_t` pointer is created called `point`, along with an integer called `sigilNum`. The function starts by looping over the list of all the entities in the level, held in the `g_entities` array (a technique you have used before). For each iteration of the loop, the current entity (which is assigned to `point`) is checked to see whether it is not in use by the game. If it isn't, the loop skips to the next entity. If the entity passes the first check, however, a comparison is made between the entity's `classname` variable and the string `team_DTF_sigil`, which you should recognize as the sigil's `classname`, as assigned in the `bg_itemlist` array.

If the strings match, a sigil entity was found, so the `teamgame.sigil[sigilNum]`'s entity is assigned to the current entity, held in `point`. Because `sigilNum` begins its life in this function as 0, this assignment would equate to the initialization of the first index of the sigil array in `teamgame`. The `sigilNum` variable is then incremented, and checked to see if it is equal to 2 (which would be the third and final sigil). If it is, the function returns to its caller; otherwise, the entity loop continues, looking for the remaining `team_DTF_sigil` items.

You now have a way of setting up the sigils at the beginning of a team game! Because `Init_Game` is called higher up in the file `g_team.c`, you will need to prototype the function. Go ahead and scroll back up to line 29, where `Team_SetFlagStatus` is prototyped, and add a declaration of `Init_Game` below it, like so:

```
void Team_SetFlagStatus( int team, flagStatus_t status );
void Init_Sigils( void );
```

## Warning *Q3* about Sigils (Or Lack Thereof)

The next test of the new sigils in your mod will be in a function called `G_CheckTeamItems`. This function is called from `G_InitGame`, one of the very first functions run by *Q3* when a new game starts up. In `G_CheckTeamItems`, specific team-related variables are initialized. Then, based on the type of game being played, certain sanity checks are performed on the most important items. For example, in a game of *CTF*, there must be a red flag and a blue flag in the map. If for some reason a non-*CTF* map were to be loaded into *Q3* when the game was expecting to launch *CTF*, *Q3* would need to warn the user that no flags exist in that map.

Head over to `g_items.c` and scroll to line 696, where the definition of `G_CheckTeamItems` begins:

```
/*
=====
G_CheckTeamItems
=====
*/
void G_CheckTeamItems( void ) {

    // Set up team stuff
    Team_InitGame();

    if( g_gametype.integer == GT_CTF ) {
        gitem_t *item;

        // check for the two flags
        item = BG_FindItem( "Red Flag" );
        if ( !item || !itemRegistered[ item - bg_itemlist ] ) {
            G_Printf( S_COLOR_YELLOW "WARNING: No team_CTF_redflag in
map" );
        }
        item = BG_FindItem( "Blue Flag" );
```

```

        if ( !item || !itemRegistered[ item - bg_itemlist ] ) {
            G_Printf( S_COLOR_YELLOW "WARNING: No team_CTF_blueflag in
map" );
        }
    }
}

```

Near the top of the function, a familiar call is made (weren't we just messing around in `Team_InitGame?`) to prepare team-related information. Then, the `g_gametype` Cvar is queried (by accessing its integer member); if the value is found to be `GT_CTF`, then a search is performed for both the red and blue flags. Creating a temporary `gitem_t` pointer called `item` starts this task, assigning it to the value of `BG_FindItem`, with an input parameter of the string "Red Flag" for the red flag and "Blue Flag" for the blue.

If the `item` pointer is `NULL` after the call to `BG_FindItem`, then the item was not found. There is one more chance for the check to be successful, however, and that is if the particular item flagged is true in the `itemRegistered` array. `itemRegistered` is a global `qboolean` array that contains either `qtrue` or `qfalse` for each item in the `bg_itemlist` array, which can be set via `RegisterItem`, a function that tells *Q3* to precache an item, regardless of whether it was found in a level or not.

If the `item` pointer is `NULL` and is not set to be precached in the `itemRegistered` array, the item was not found anywhere in the level, and so the player is warned. The warning comes in the form of a call to `G_Printf`, which simply prints a string of text to the console. The console, if you remember, can be toggled on and off via the tilde (~) key. You can specify the color of the warning with the first parameter; in the case of the missing *CTF* flags, the warning text is written in yellow (as indicated by the `S_COLOR_YELLOW` flag).

### TIP

The string-color flags are declared on line 511 of `q_shared.h`, and allow you to use any one of eight different colors for your specific text. They are fairly self-explanatory; `S_COLOR_BLACK` is black, `S_COLOR_RED` is red, and so on.

To add the appropriate checks for your `GT_DTF` game type, add the following code after the `GT_CTF` section in `G_CheckTeamItems` ends, like so:

```

        G_Printf( S_COLOR_YELLOW "WARNING: No team_CTF_blueflag in
        map" );
    }
}

if ( g_gametype.integer == GT_DTF )
{
    gitem_t    *item;

    // check for at least one sigil
    item = BG_FindItem( "Flag" );
    if ( !item || !itemRegistered[item - bg_itemlist] )
        G_Printf( S_COLOR_YELLOW "WARNING: No team_DTF_sigil in
        map" );
}

```

In this code snippet, you can see that a single check is performed for one sigil. As with the *CTF* flag check before it, the "Flag" string is passed to `BG_FindItem` (remember that you had the display name of the sigil actually read "Flag"), and the result is saved in `item`. If `item` is `NULL`, and its indexed value is `qfalse` in the `itemRegistered` array, a warning is printed to the console telling the player to get the heck out of the level; there is no `team_DTF_sigil` item type found in the map.

## NOTE

Although there will be three sigils in *DTF*, technically, only one is really needed. You could easily modify this function to loop three times, keeping a counter for the current total of found sigils, and if the final total wasn't `MAX_SIGILS`, you could throw your warning message up to the console.

## Creating the New Sigil Behavior

You have touched the sigil with the finger of God (so to speak) and created it from the dust of so many bits and bytes. It has a new identifying `classname` that distinguishes it from all the previous *Q3* items,



and has absorbed the visual aspects of the *CTF* flag. It is now time to take your new item to obedience school so that it can learn new behaviors. As I have mentioned, flags in *CTF* can be picked up, carried, or dropped, or can remain idle in their team's base. *DTF* sigils, on the other hand, will never be picked up, carried, or dropped in the middle of a map. They will, however, change colors when they are touched, and automatically generate a score for the team that holds them. This section covers the necessary code to implement these behaviors.

## Getting Touchy-Feely with `Touch_Item`

All items in *Q3* can be touched. In fact, as the player runs around the map, attacking opponents and fighting for survival, he almost perpetually comes into contact with items. Touching weapons adds them to the player's inventory; touching a powerup will grant that power to the player for a limited time. It goes without saying that touching a *CTF* flag will cause the flag to be picked up (if it is the enemy flag) or returned home (if it is the team flag, having been dropped in the map). There is a function that controls what each item will do if it is touched, and it is aptly named `Touch_Item`.

`Touch_Item`'s definition can be found near line 393 of `g_items.c`. Take a look at the first part of the function body:

```
void Touch_Item (gentity_t *ent, gentity_t *other, trace_t *trace) {
    int          respawn;
    qboolean     predict;

    if (!other->client)
        return;
    if (other->health < 1)
        return;          // dead people can't pickup

    // the same pickup rules are used for client side and server side
    if ( !BG_CanItemBeGrabbed( g_gametype.integer, &ent->s,
        &other->client->ps ) ) {
        return;
    }
}
```

The `Touch_Item` function takes two `gentity_t` pointers as input, one called `ent` and one called `other`. In the context of this function, `ent` refers to the item that was touched, and `other` refers to the player that touched said item. A third pointer, a `trace_t` called `trace`, is also passed into the function, but is presently unused, so you don't need to worry about it.

The first few lines of the function check to see whether the player who touched the item (referenced by `other->client`) is a valid player, and that he is not dead (because dead players do not need to pick up items). If the first two checks pass, a call is then made to a function named `BG_CanItemBeGrabbed`. This function performs a series of sanity checks on a specified item to see if the player in question can actually pick it up.

Take a look at the definition for `BG_CanItemBeGrabbed` on line 1063 of `bg_misc.c`. The first part of it reads as follows:

```
qboolean BG_CanItemBeGrabbed( int gametype, const entityState_t *ent,
const playerState_t *ps ) {
    gitem_t *item;
#ifdef MISSIONPACK
    int      upperBound;
#endif

    if ( ent->modelindex < 1 || ent->modelindex >= bg_numItems ) {
        Com_Error( ERR_DROP, "BG_CanItemBeGrabbed: index out of range" );
    }

    item = &bg_itemlist[ent->modelindex];

    switch( item->giType ) {
    case IT_WEAPON:
        return qtrue;    // weapons are always picked up

    case IT_AMMO:
        if ( ps->ammo[ item->giTag ] >= 200 ) {
            return qfalse;    // can't hold any more
        }
        return qtrue;
    }
```

`BG_CanItemBeGrabbed` starts by first looking at the `modelindex` member of `entityState_t` in the item. The `modelindex` member of `entityState_t` is an integer that points to one of the elements in the `bg_itemlist` array; its range can be anywhere from 1 to `bg_numItems`. And, although I have reiterated that arrays begin at element 0 in C, this first element is unusable, since it is reserved in the `Q3` code with NULL values; hence, the range begins at 1. If the value found in `modelindex` is invalid, `BG_CanItemBeGrabbed` throws an error in `Q3`, causing the game to stop loading.

Next, a local `gitem_t` pointer called `item` is assigned to represent the actual item passed into the function (done by referencing its place in the `bg_itemlist` array). Then, a switch block is entered, based on the item's `giType`. Remember that the `giType` property of a `gitem_t` variable holds the flag that identifies the item as belonging to a certain category of items. The first two shown in this code snippet are `IT_WEAPON` (which can always be picked up) and `IT_AMMO` (which can always be picked up, as long as the player's current ammunition count isn't beyond 200).

Every category of item is listed in this switch block, including `IT_TEAM`, which is the category assigned to the `CTF` flags. You will need to add your sigil's category, `IT_SIGIL`, to this switch block, and create logic that will tell `Q3` whether your sigil can be picked up. Your logic should be simple and straightforward.

- Are you a red team member, touching a sigil that is red? If so, you are disallowed.
- Are you a blue team member, touching a sigil that is blue? If so, you are disallowed.
- Otherwise, you are allowed to touch the sigil.

### TIP

The error-printing function, `Com_Error`, uses a special type of flag as its first input parameter: that of an `errorParm_t` enum, which can be found declared on line 381 of `q_shared.h`. The two scariest types of errors to throw are `ERR_FATAL`, which causes `Q3` to exit completely, and `ERR_DROP`, which prints the error to the screen and disconnects the player from the active game session.

Scroll down to line 1201, while still in the same file, and add the following code right after the IT\_TEAM's case statement completes:

```

case IT_SIGIL:
    // red team cannot touch a red sigil
    if (ps->persistant[PERS_TEAM] == TEAM_RED && ent->powerups ==
        PW_SIGILRED)
        return qfalse;
    // blue team cannot touch a blue sigil
    else if (ps->persistant[PERS_TEAM] == TEAM_BLUE &&
        ent->powerups == PW_SIGILBLUE)
        return qfalse;
    else
        return qtrue;

```

In this snippet of code, the current team of the player in question is referenced, with a call to `ps->persistant[PERS_TEAM]`. You have not yet handled the logic that assigns a sigil a particular color. When you look at this code snippet, you'll see that the cat is out of the bag: The `powerup_t` value will be held in `ent->powerups`. I will get to the actual assignment of colors soon; for now, use this code with that assumption in place. As the rules for touching the sigils dictate, if the player's team is red, and the sigil is red, `qfalse` is returned. If the player's team is blue, and the sigil is blue, `qfalse` is also returned. For all other combinations of values, `qtrue` is returned, which allows for mixed color/team touching (that is, a member of the red team can touch a blue sigil, and so on), as well as white sigil touches (either team may touch a white sigil).

## Wiring Sigil\_Touch for Touch\_Item

Now that you have logic handling whether a specific sigil can be touched, let's revisit `Touch_Item` to see what happens next. A few lines down from the call to `BG_CanItemBeGrabbed` is the start of another switch block, based on the category of the item:

```

// call the item-specific pickup function
switch( ent->item->giType ) {
case IT_WEAPON:
    respawn = Pickup_Weapon(ent, other);
    break;

```

```

case IT_AMMO:
    respawn = Pickup_Ammo(ent, other);
    break;

```

In each case statement, an integer called `respawn` is assigned the value of a specific function that handles each item category. For these first two item categories, the `IT_WEAPON` type makes a call to `Pickup_Weapon`, while the `IT_AMMO` type calls `Pickup_Ammo`. Each of these functions passes in the item (`ent`) and the player touching the item (`other`). As expected, the `IT_TEAM` category is also present in this, and it calls `Pickup_Team`. `Pickup_Team` breaks up the logic for team-related items, and disperses it to other functions for appropriate updates. In a nutshell, `Pickup_Team` performs the following tasks for *CTF*:

- It finds out what team owns the touched flag.
- If the player's team matches the touched flag (for example, a member of the red team touches a red flag), it either sends the flag home (if it has been dropped) or scores a capture (if the player is carrying the enemy's flag). It then processes the appropriate scoring.
- If the player's team does not match the touched flag (for example, a member of the red team touches a blue flag), it adds the flag powerup to the player and updates the flag's status (`FLAG_TAKEN`). It then processes the appropriate scoring.

Sounds easy enough, right? You should be able to implement a function that performs a similar set of tasks for the *DTF* sigils. Start near line 444 of `g_items.c`, right after the `IT_TEAM` case is handled in `Touch_Item`, and add a new case like so:

```

case IT_TEAM:
    respawn = Pickup_Team(ent, other);
    break;
case IT_SIGIL: // new dtf sigil type
    respawn = Sigil_Touch(ent, other);
    break;

```

Very simply, you enter the case for an item discovered to be of type `IT_SIGIL`, and perform the `respawn` assignment by calling a new function called `Sigil_Touch`. Your next job will be to create that function. Place `Sigil_Touch` on line 944 of `g_team.c`, because the other pickup-related functions are declared in that file as well:

```

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

    if (!cl)
        return 0;

    if (ent->count && ent->nextthink < level.time + 1500) //
protect against overflows by not counting
        return 0;

    if ( cl->sess.sessionTeam == TEAM_RED && ent->s.powerups !=
PW_SIGILRED )
    {
        ent->nextthink = level.time - (level.time % 4000) + 4000;
        ent->think = Sigil_Think;
        ent->s.powerups = PW_SIGILRED;
        ent->s.modelindex = ITEM_INDEX( BG_FindItemForPowerup(
PW_SIGILRED ) );
        ent->count = 1;
    }
    else if ( cl->sess.sessionTeam == TEAM_BLUE && ent->s.powerups !=
PW_SIGILBLUE )
    {
        ent->nextthink = level.time - (level.time % 4000) + 4000;
        ent->think = Sigil_Think;
        ent->s.powerups = PW_SIGILBLUE;
        ent->s.modelindex = ITEM_INDEX( BG_FindItemForPowerup(
PW_SIGILBLUE ) );
        ent->count = 1;
    }
    return 0;
}

```

This is a hefty function; let's take it one step at a time. `Sigil_Touch` begins with a sanity check to the player, to make sure it is valid. The

next line, which checks `ent->count` and `ent->nextthink`, is performed to prevent players from constantly touching sigils . . . an event that could call functions unnecessarily and consume resources (also referred to as *overflowing*).

If the checks pass, then the real logic begins to determine what will happen when a specific sigil is touched. The code basically mirrors itself; a blue team member will set the sigil's `s.powerups` value to `PW_SIGILBLUE`, while a red team member will set the sigil's `s.powerups` value to `PW_SIGILRED`. You should also note that the sigil's `s.modelindex` value is set to the return of `BG_FindItemForPowerup`, passing in the new sigil color (`PW_SIGILBLUE` or `PW_SIGILRED`).

The common assignments that take place, regardless of team or sigil color, involve timing, setting the count value, the sigil's think function, and nextthink time. The assignment of `ent->count` to 1 reflects the earlier statement made about overflows; by setting this value to 1, you can prevent an immediate re-touch of the sigil, causing excess calls to the `Sigil_Touch` function. The sigil will need to have a think function once a team holds it, so the assignment of `ent->think` to `Sigil_Think` performs this action (you will write `Sigil_Think` shortly).

### TIP

`ITEM_INDEX` is a macro that is wrapped around the call to `BG_FindItemForPowerup`. It is declared on line 639 of `bg_public.h`, and simply returns a numeric value that represents the found item's place in the global `bg_itemlist` array.

Finally, because the sigil will be running a think function, you will need to tell the sigil the next valid time it can run said function. Because you will want all sigils to perform their think function at a specific interval, you will need to use an algorithm to set the nextthink value to a specific time, regardless of the current game time. That algorithm is as follows, where  $x$  is a proper nextthink value that always calls the think function every two seconds, maintaining this touched sigil's synchronization with other sigils:

$$x = \text{current time} - (\text{current time} \% 4000) + 4000$$

The last thing you will need to take care of is prototyping `Sigil_Touch`, because it is called from `Touch_Item` in `g_items.c`. Open `g_team.h` and,

at the very end of the file after the declaration of `Pickup_Team`, add the following line of code:

```
int Sigil_Touch( gentity_t *ent, gentity_t *other ); // dtf
```

Perfect! `Sigil_Touch` is now ready to go.

## Keeping Score with `Sigil_Think`

In the previous section, when you dealt with handling a sigil that had been touched, you saw that the sigil was assigned a think function called `Sigil_Think`. That's because when a sigil is held by a team in a game of *DTF*, that team will start accumulating points based on how many sigils are held. Each sigil should assign a capture point of 1 for every four seconds that pass in the game. Just above the `Sigil_Touch` function in `g_team.c`, add the following code for `Sigil_Think`:

```
/*
=====
Sigil_Think
=====
*/
void Sigil_Think( gentity_t *ent ) {
    team_t team;

    team = (ent->s.powerups == PW_SIGILRED) ? TEAM_RED : TEAM_BLUE;
    ent->count = 0;
    level.teamScores[team]++;
    ent->nextthink = level.time + 4000;

    //refresh scoreboard
    CalculateRanks();
}
```

`Sigil_Think` is relatively small, starting with the declaration of a local `team_t` variable called `team`. Then, the value of `team` is assigned either `TEAM_RED` or `TEAM_BLUE`, based on whether the passed-in entity (the sigil itself) is colored red or blue, respectively. Next, the sigil's count property is reset to 0, marking the sigil safe to be touched again by players. The team scores are held in a global variable, `level.teamScores`, which



is actually an array that has two indexes, either `TEAM_RED` or `TEAM_BLUE`. Because your local team variable is set earlier and then used as the index in the score addition, the appropriate team's score will reflect the change. The sigil's `nextthink` property is then set to 4000 milliseconds from the current game time, which will repeat this entire process again every four seconds.

A concluding function call named `CalculateRanks` is made at the end of `Sigil_Think`. This function is simply provided to update the scoreboard when an important event takes place. Sigils increment the teams' scores every four seconds, so this qualifies as an important-enough event to refresh the player's scoreboard. With both `Sigil_Touch` and `Sigil_Think` in place, you can now return to `Touch_Item` to clean up any leftover code. As it turns out, there is one more place in the `Q3` code base that needs updating.

## Keeping cgame in Check

When standard *CTF* flags are picked up, they play a default sound effect. This sound effect is handled on the client side of things, safely tucked away in `cgame`. Because the *DTF* sigils have borrowed the *CTF* flag's default sounds, changes need to be made in both `game` and `cgame` to support the sigils playing these sound effects. You can start by heading back to `g_items.c` and scrolling down to around line 465, in the heart of `Touch_Item`. Here, you should see code that looks like the following snippet:

```
// play the normal pickup sound
    if (predict) {
        G_AddPredictableEvent( other, EV_ITEM_PICKUP, ent->s.modelindex
);
    } else {
        G_AddEvent( other, EV_ITEM_PICKUP, ent->s.modelindex );
    }
// powerup pickups are global broadcasts
    if ( ent->item->giType == IT_POWERUP || ent->item->giType ==
IT_TEAM ) {
        // if we want the global sound to play
        if (!ent->speed) {
            gentity_t *te;
```

```

te = G_TempEntity( ent->s.pos.trBase,
EV_GLOBAL_ITEM_PICKUP );
te->s.eventParm = ent->s.modelindex;
te->r.svFlags |= SVF_BROADCAST;

```

In this segment of code, a variable called `predict` is checked for true or false. Because it is false by default, and you do not explicitly set it to true for `IT_SIGIL`, `G_AddEvent` is called, passing in an event flag of `EV_ITEM_PICKUP`. This will create a client-side event to play a “pickup” sound on the client, and will do so for all items, including your new sigils. Next, the item category is checked for a value of either `IT_POWERUP` or `IT_TEAM`. If either is found, the code continues to play a global pickup sound to all players. You don’t want your sigil left out in the cold, so go ahead and modify that line so it reads as follows:

```

if ( ent->item->giType == IT_POWERUP || ent->item->giType ==
IT_TEAM || ent->item->giType == IT_SIGIL ) {

```

Now, when a sigil is touched, the game code will send a signal to `cgame`, letting it know that it needs to play a generic global pickup sound effect, which all players will hear.

To actually have `cgame` play that sound effect, open `cg_event.c` and scroll down to line 666, which puts you smack-dab into `CG_EntityEvent` (remember him from Chapter 6, “Client Programming”?). All events that are created by entities will be handled in this function. At this position in the file, you should be within the case for `EV_ITEM_PICKUP`. Find the line of code that reads like this:

```

if ( item->giType == IT_POWERUP || item->giType == IT_TEAM ) {

```

and change it to the following:

```

if ( item->giType == IT_POWERUP || item->giType == IT_TEAM || item-
>giType == IT_SIGIL ) {

```

## NOTE

**This function actually continues deeper into `Touch_Item`, creating the identical `EV_GLOBAL_ITEM_PICKUP` event, but plays it locally to a client if the player’s speed value is not 0. You can see this in the snippet earlier with the line of code that starts with `if (!ent->speed)`.**

This allows the `cgame` code to properly interpret the `EV_ITEM_PICKUP` event for sigils and play the default sound (which, coincidentally, is the sound used to pick up a health pack), held in `cgs.media.n_healthSound`.

You're creating some exciting stuff here! With the sigils identified and uniquely described within `Q3`, they can now think and react when players touch them, changing colors and incrementing team scores over time. The only thing you need to do now is get the actual sigil spawned in a level. The next section discusses how to dynamically create sigil spawn points on the fly, without having to recompile any maps.

## Tricking `Q3`: Reusing Spawn Points

Every year, id Software hosts a giant party in Texas for the many fans of *Quake*. This gaming fest is referred to as *QuakeCon*. Back at QuakeCon '97, volunteers were in the midst of trying to quickly come up with a way to rewrite code in QuakeC to allow for more players spawning into maps. Because QuakeCon has typically held a tournament each year, 1997 was no different. However, the code had not been prepared ahead of time, and anxious players were getting annoyed at the delay. The volunteers who were modifying QuakeC could not get the new tournament mod to work properly; every time they had a new release, players would try it, only to find that many of them were spawning into each other, which of course, meant a *telefrag* and instant death. There simply weren't enough spawn points in the selected tournament maps to allow for so many players.

Someone stepped out of the crowd and offered to help. He sat down and began to look at what could be done to solve the problem as fast as possible. His answer? Dynamically turning the spawn points for health packs into player spawn points. His trick worked, and the tournament got started, allowing many players to spawn into maps with only a few starting points to begin with. The person who solved the problem was none other than Dave Kirsch, the same man responsible for *CTF*. The technique he used of changing level entities on the fly has been reused many times since then, and offers mod programmers an additional level of flexibility. Now, new entity types no longer have to be dictated to level designers ahead of time; mod programmers can

simply re-map old entities to new ones as the level loads. In this next section, you'll apply this technique in several formats to create *DTF*-ready levels, without having to recompile the maps from scratch.

## The Process of Spawning Level Entities

In order to attempt such a feat as changing level entities, you must first understand how they are handled normally. When *Q3* fires up a new level, it runs through a series of initialization functions (you saw one of these earlier, `G_CheckTeamItems`). The *Q3* engine parses out the physical map file, and the level entities are returned to the game code via the use of a system-call function named `trap_LocateGameData`. This commits the appropriate entity information into a global variable called `level.gentities` (which itself is a pointer to the `g_entities` array). And, you should know by experience that every single entity in *Q3* is held in the `g_entities` array.

Once the level data is properly processed into the game code, the server-side DLL takes over and begins preparing each entity to be launched into the map. It does this by first making a call to `G_SpawnEntitiesFromString`, which can be found on line 665 of `g_spawn.c`:

```
void G_SpawnEntitiesFromString( void ) {
    // allow calls to G_Spawn*()
    level.spawning = qtrue;
    level.numSpawnVars = 0;

    // the worldspawn is not an actual entity, but it still
    // has a "spawn" function to perform any global setup
    // needed by a level (setting configstrings or cvars, etc)
    if ( !G_ParseSpawnVars() ) {
        G_Error( "SpawnEntities: no entities" );
    }
    SP_worldspawn();

    // parse ents
    while( G_ParseSpawnVars() ) {
        G_SpawnGEntityFromSpawnVars();
    }
}
```

## 240 7. Defend the Flag

```
    level.spawning = qfalse;                // any future calls to
G_Spawn*() will be errors
}
```

As demonstrated by this function, a check is first made to see if there are actually any entities loaded that can be spawned. If not, an error is thrown with a call to `G_Error`. Otherwise, the world is created through a call to `SP_worldspawn`. Then, all the level entities are parsed and spawned into the map through a call to `G_ParseSpawnVars`, which places the entity strings for each item parsed out of the level data into a global array called `level.spawnVars`. Once this global array is populated, `G_SpawnGEntityFromSpawnVars` takes over.

In the function `G_SpawnGEntityFromSpawnVars` (found on line 449 of `g_spawn.c`), each entity string is looked at in order in the `level.spawnVars` array. As each entity string is evaluated, a physical entity is prepared by `Q3` to hold the parsed-out entity (via a call to `G_Spawn`). The entity string's name and value are obtained from `level.spawnVars` and placed into the physical entity, performed by a call to `G_ParseField`. For example, the entity string's name might be `classname`, and its value might be `team_CTF_redflag`. A series of rules are then applied, which allow `Q3` to specifically refrain from spawning a particular entity based on a series of special entity strings that can be added by the level designer. For example, if the `notteam` string is found (and its value is 1), then the entity being handled will not be spawned into `Q3` in team-based games.

Finally, after all these checks have been performed, the final physical entity, which has had all the current level entity's info passed into it, is spawned into the map through a call to `G_CallSpawn`, which can be found near line 252 of `g_spawn.c`:

```
qboolean G_CallSpawn( gentity_t *ent ) {
    spawn_t      *s;
    gitem_t      *item;

    if ( !ent->classname ) {
        G_Printf ( "G_CallSpawn: NULL classname\n" );
        return qfalse;
    }

    // check item spawn functions
```

## Stringing Entities Along

Entity strings are simply the key/value pairs that represent specific information about level entities that are encoded within a map's BSP file. All level entities that are placed in a BSP by a level designer are parsed out as a set of curly-braced names and values. For example, a set of entity strings for a player's deathmatch spawn point might look like this:

```
{
"classname" "info_player_deathmatch"
"angle" "360"
"origin" "432 -208 32"
}
```

In this set of entity strings, the `classname` is `info_player_deathmatch`, its angle in the world is 360, and its location in the world is  $432 \times -208 \times 32$ . All three entity strings belong to the same entity (`info_player_deathmatch`) as indicated by the opening and closing curly braces.

```
for ( item=bg_itemlist+1 ; item->classname ; item++ ) {
    if ( !strcmp(item->classname, ent->classname) ) {
        G_SpawnItem( ent, item );
        return qtrue;
    }
}

// check normal spawn functions
for ( s=spawns ; s->name ; s++ ) {
    if ( !strcmp(s->name, ent->classname) ) {
        // found it
        s->spawn(ent);
        return qtrue;
    }
}

G_Printf ("%s doesn't have a spawn function\n", ent->classname);
```

```

    return qfalse;
}

```

`G_CallSpawn` first looks at the `ent->classname` of the item entity that was passed to it. If the `classname` is empty, the function returns with an error. Otherwise, the function moves on to begin looping through the `bg_itemlist` array (starting at index 1 instead of 0, because, as I mentioned earlier, the first index of the array is reserved and never used by a valid item). For every item in `bg_itemlist` that has a valid `classname`, a comparison is made between the `classname` found in the array and the `classname` of the current item. If a match is discovered, `G_SpawnItem` creates the entity and places it in the map, saving its entity-specific information found in the `bg_itemlist` array.

A follow-up loop is handled that loops over the `spawns` array, dealing with the setup of such things as player spawn points and triggers—valid entities in the map that aren't necessarily associated with models, sound effects, images, and the like. If `G_CallSpawn` makes it all the way to the end of the function and has still not found an appropriate spawn function for the entity that was passed to it, it alerts the player with a message that the entity has no spawn function, and the function exits.

## Jimmying Item Entities into a Map

Because *DTF* will utilize the new `team_DTF_sigil` item entity, you need to find a way to call its spawn function in `G_CallSpawn`. The trick is that the maps will already be compiled, and their level entities will be in a complete and unmodifiable state. The solution is to programmatically find appropriate item entities, and replace them with the `team_DTF_sigil` item. Because *DTF* will be played much like *CTF* (and use the same models as the *CTF* flags), it makes sense that the first item entities to overthrow will be `team_CTF_redflag` and `team_CTF_blueflag`.

The first step in this process will be to properly register the three item entity states that a sigil can be during a game of *DTF*. If you recall, when you made the addition of the sigils to `bg_itemlist`, you created three entries: `team_DTF_sigil`, `team_DTF_sigil_red`, and `team_DTF_sigil_blue`. These item entities were also assigned `powerup_t` flags to uniquely identify them, as `PW_SIGILWHITE`, `PW_SIGILRED`, and

PW\_SIGILBLUE, respectively. Remember when you were looking at `G_CheckTeamItems`, and I explained that an item could be registered by setting its location in the `itemRegistered` array to true? Well, because pre-existing *CTF* maps will not have the *DTF* sigils in them, this will be the technique you will use.

The first thing you will need to do is call `RegisterItem` within `G_CallSpawn` to prepare the three sigil item types for spawn processes. This will properly flag the three sigil entities in the `itemRegistered` array; that way, *Q3* will know how to handle them, even if they don't already exist in the map being loaded. Right after the check to see if `G_CallSpawn`'s passed-in `ent` variable has a valid classname, make the following code adjustment:

```

    G_Printf ("G_CallSpawn: NULL classname\n");
    return qfalse;
}

if (g_gametype.integer == GT_DTF)
{
    RegisterItem(BG_FindItemForPowerup(PW_SIGILWHITE));
    RegisterItem(BG_FindItemForPowerup(PW_SIGILRED));
    RegisterItem(BG_FindItemForPowerup(PW_SIGILBLUE));
}

```

Here, `g_gametype.integer` is checked for a value of `GT_DTF`, the flag that indicates your *Defend the Flag* game type. If *DTF* is detected, all three sigil item states (white, red, and blue) are flagged in the `itemRegistered` array with a call to `RegisterItem`, using the index that is returned from `BG_FindItemForPowerup`. This new function, `BG_FindItemForPowerup`, traverses the entire `bg_itemlist` array, looking for a match to the passed-in item entity. It's defined near line 957 in `bg_misc.c`:

```

gitem_t    *BG_FindItemForPowerup( powerup_t pw ) {
    int      i;

    for ( i = 0 ; i < bg_numItems ; i++ ) {
        if ( (bg_itemlist[i].giType == IT_POWERUP ||
              bg_itemlist[i].giType == IT_TEAM ||
              bg_itemlist[i].giType == IT_PERSISTANT_POWERUP ) &&

```



```

        bg_itemlist[i].giTag == pw ) {
            return &bg_itemlist[i];
        }
    }

    return NULL;
}

```

As expected, the function takes a passed-in `powerup_t` variable called `pw`, and begins stepping through the entire `bg_itemlist` array, comparing `pw` against the `giTag` found in the current iteration through the loop, returning the found item when a match is discovered. By looking at this code, however, you'll notice that only the `IT_POWERUP`, `IT_TEAM`, and `IT_PERSISTANT_POWERUP` categories are even considered. The `IT_SIGIL` flag categorizes the *DTF* sigils, so you will need to add this `giType` check as well. Modify the `if` statement within the `for` loop, so that it reads as follows:

```

        if ( (bg_itemlist[i].giType == IT_POWERUP ||
            bg_itemlist[i].giType == IT_TEAM ||
            bg_itemlist[i].giType == IT_PERSISTANT_POWERUP ||
            bg_itemlist[i].giType == IT_SIGIL ) &&
            bg_itemlist[i].giTag == pw ) {

```

Excellent. With the addition of the `IT_SIGIL` flag to the `giType` validation, the new *DTF* sigils will be included in the scan for a `powerup` item match.

## Yanking Out CTF Flags

Now that the *DTF* sigils are valid registered item entities within *Q3*, it's time to pull the ol' switcheroo on existing items. Jump back to `G_CallSpawn` in the `g_spawn.c` file. As mentioned earlier, near line 368, a loop begins over the `bg_itemlist` array, looking for the proper item that matches the entity passed to `G_CallSpawn`. This will be where you hijack the *CTF* flags and convert them to *DTF* sigils. Make the following changes to that loop:

```

    // check item spawn functions
    for ( item=bg_itemlist+1 ; item->classname ; item++ ) {
        if ( !strcmp(item->classname, ent->classname) ) {

```

```

// if a CTF flag is found, and the game type is DTF
// convert the flag point to a sigil point
if ( item->giType == IT_TEAM && g_gametype.integer ==
GT_DTF) {
    item = BG_FindItemForPowerup(PW_SIGILWHITE);
    ent->classname = item->classname;
}

G_SpawnItem( ent, item );
return qtrue;
}
}

```

This code demonstrates how a *CTF* flag point conversion is achieved. After the `item->classname` matches the value found in `ent->classname`, a check is performed on the item's `giType`. If the `giType` of the matched item is `IT_TEAM` (the category to which the *CTF* flags are assigned), and the `g_gametype.integer` is found to be `GT_DTF`, the matched item is assigned a new value—that of the `team_DTF_sigil` item entity, as referenced by the `PW_SIGILWHITE` powerup\_t flag. The entity's `classname` is then overwritten by the new item's `classname` (`team_DTF_sigil`). At this point, the entity passed to `G_CallSpawn` is no longer the entity parsed from the map's entity strings. The item retains all the original entity's information, such as location and angle, as found in the map, but because its `classname` is converted to `team_DTF_sigil` and it obtains all the other values of the `team_DTF_sigil` item entity (such as `IT_TEAM` for the `giType`), the result is one brand spanking new `team_DTF_sigil` item entity. This is then spawned into the map with `G_SpawnItem`. And, because you know `IT_TEAM` will come up twice (once for each *CTF* flag), the result of this code will be two new sigil points.

## Creating a Third Sigil Spawn Point

As with all *CTF* maps, there are two flag spawn points: one for the red flag, and one for the blue. Additionally, any maps that are built for the Team Arena Expansion Pack have a third flag spawn point: one for the white flag. Conveniently, all three flags are categorized in the `IT_TEAM` `giType`. You won't, however, always have the benefit of using a Team Arena map; players will ultimately want to use standard *CTF*

maps when they play your mod. So, you will need to come up with a way to dynamically generate a third spawn point for your last sigil.

In order to come up with an algorithm that will help you determine where to create the third spawn point, you need to decide on a set of rules to use. Based on what you already know about the layout of a *CTF* map, and what your objective will be for a final *DTF* conversion, these are the rules that will apply:

- If three sigils aren't already detected, a third sigil spawn point will be generated.
- The third sigil spawn point should find the general center point between the first two sigils, and look in that area for an existing entity to take over.
- Existing entities can be armor or health packs.
- If no health packs or armor are found in the level, a weapon spawn point will be picked as a last resort (because every map must have a weapon somewhere).

Now that you have rules in place to generate a third sigil, let's see what it will take to implement those rules in a function.

First, take a step back to `G_SpawnEntitiesFromString` in `g_spawn.c`, which you'll recall is one of the main `init` functions for a game of *Q3*. After the call to `G_ParseSpawnVars`, make the following changes:

```
// parse ents
while( G_ParseSpawnVars() ) {
    G_SpawnGEntityFromSpawnVars();
}

// make sure dtf maps have a 3rd sigil
if (g_gametype.integer == GT_DTF)
    G_ValidateSigils();
```

Here, the game type is once again verified to be `GT_DTF`; as a result, a new function called `G_ValidateSigils` will execute. The body of `G_ValidateSigils` is handled next, so go ahead and place the following function definition directly above `G_SpawnEntitiesFromString` (so it doesn't have to be prototyped):

```
void G_ValidateSigils()
{
```

```

    gentity_t    *it_ent;

    it_ent = G_Spawn();
    it_ent->think = ValidateSigilsInMap;
    it_ent->nextthink = level.time + 500;
}

```

This tiny function simply creates a new entity called `it_ent`, and spawns it into the map with `G_Spawn`. Note that the newly created entity's `think` function is explicitly set to `ValidateSigilsInMap`, a function you will write next. The final line of code tells *Q3* that this new entity will perform its `think` function exactly half a second from spawning.

To lay out `ValidateSigilsInMap`, you will need to jump back to `g_team.c`, because it will reference the `teamgame` variable that is accessible only in that file. This function is tricky and complicated, so I'll take it a few blocks of code at a time. Start by opening `g_team.c`, scrolling down to about line 207, where `Init_Sigils` ends, and begin the new function:

```

#define FRADIUS 800
void ValidateSigilsInMap( gentity_t *ent )
{
    vec3_t      start, end, temp, mins, maxs, tvec, offset = {FRADIUS,
FRADIUS, FRADIUS};
    int         numEnts, i, touch[MAX_GENTITIES], dist = FRADIUS;
    gentity_t   *tent, *targ;
    float       vlen;
    qboolean    foundItem = qfalse, foundPreferredItem = qfalse;
    gitem_t     *item;

    // if 3rd sigil exists, this function doesn't need to run
    if (teamgame.sigil[2].entity)
        return;

    VectorCopy(teamgame.sigil[0].entity->r.currentOrigin, start);
    VectorCopy(teamgame.sigil[1].entity->r.currentOrigin, end);

```

This function begins by looking at the entity pointer of the third sigil (which would be held in `teamgame.sigil[2].entity`). If it is not `NULL`, the function exits, because there is no need to create a third sigil; it

already exists. If, however, there is still the need for a third sigil, the positions of the existing two sigils are copied into two `vec3_t` variables, named `start` and `end`. These locations on the map allow you to gauge a rough determination of where the center is. The next bit of code handles that:

```
VectorSubtract(start, end, temp);
VectorScale(temp, 0.5, temp);
VectorAdd(end, temp, temp);

VectorCopy(temp, mins);
VectorCopy(temp, maxs);
VectorAdd(maxs, offset, maxs);
VectorScale(offset, -1, offset);
VectorAdd(mins, offset, mins);
```

In this snippet of code, the sigil's locations are subtracted, and the result is saved to `temp`. Next, the distance in `temp` is multiplied or *scaled* by 0.5, effectively cutting the distance in half. `VectorAdd` is then called, adding the scaled `temp` `vec3_t` back to `end` to get an “offset” value, which will be the center position between the two sigils.

After the center point has been found, an appropriate range must be created around that point. That range will be scanned for a possible candidate to use as a replacement for a sigil. You accomplish this by copying the center point vector into `mins` and `maxs`, then creating a positive and negative offset, equal to the size of a pre-defined range held in `FRADIUS`. As you can see, the outer limit of the range is created by the call to `VectorAdd`, adding the offset vector to `maxs`, while the inner limit of the range is created by adding the offset to `mins` *after* offset has been made negative by multiplying it by `-1`.

After you have the range, you can make a call to `trap_EntitiesInBox` to look for a possible valid entity, the very same function you used way back in Chapter 3, “More Weaponry Work,” for the gravity well:

```
numEnts = trap_EntitiesInBox( mins, maxs, touch, MAX_GENTITIES );
for ( i=0 ; i<numEnts ; i++ )
{
    tent = &g_entities[touch[i]];

    if (!tent->item)
```

```
        continue;

        if (!(tent->item->giType == IT_HEALTH || tent->item->giType ==
IT_ARMOR || tent->item->giType == IT_WEAPON))
            continue;

        VectorSubtract(temp, tent->r.currentOrigin, tvec);
        vlen = abs(VectorLength(tvec));

        if (vlen > FRADIUS)
            continue;
```

When `numEnts` is assigned the result of `trap_EntitiesInBox`, the loop can begin, looking at every entity that was found in the check. If the entity is not in use, it can't be converted. If the entity is not an `IT_HEALTH`, `IT_ARMOR`, or `IT_WEAPON` item type, it cannot be converted.

The next bit of code improves upon the technique used back in the gravity-well tutorial. A new distance vector, `tvec`, is assigned the difference between the current entity's origin and the center point of the range (still held in `temp`). Because `tvec` can be positive or negative at this point, the length of the vector is determined by a call to `VectorLength`, and the result is passed to the `abs` function. This returns the positive value for a positive or negative number. This final value is saved to `vlen`.

## Picking the Preferred Item

The next bit of code is the most complicated part, so I'll take it slow and cover each detail carefully:

```
        if ( (foundItem && !foundPreferredItem) &&
            (tent->item->giType == IT_HEALTH || tent->item->giType ==
IT_ARMOR) ) {
            foundPreferredItem = qtrue;
            dist = abs(VectorLength(tvec));
            targ = tent;
        } else {
            if ( vlen < dist ) {
                if (tent->item->giType == IT_HEALTH ||
                    tent->item->giType == IT_ARMOR ||
```

## 250 7. Defend the Flag

```
        (tent->item->giType == IT_WEAPON &&
!foundPreferredItem) ) {
        foundItem = qtrue;
        dist = abs(VectorLength(tvec));
        targ = tent;

        if (tent->item->giType == IT_HEALTH || tent-
>item->giType == IT_ARMOR)
            foundPreferredItem = qtrue;
    }
}
}
```

In the scan of the entities found, you would like to convert a health pack or armor into the third sigil. These are what you will refer to as the *preferred item*. If, however, you cannot find either within the range, you will have to fall back on a simple weapon as your replacement. For each iteration through the loop of entities, you will check two flags with which you monitor these findings: `foundItem` and `foundPreferredItem`.

In the first part of this code snippet, you check to see if you have already found a non-preferred item (in a previous iteration through the loop). If this is true, look at the current entity in the loop and see if it is either a health pack (`IT_HEALTH`) or armor (`IT_ARMOR`). If it is, you have found your preferred item; set the `foundPreferredItem` flag to true, update the `dist` variable to hold the absolute distance of the `tvec` `vec3_t`, and assign `tent` to `targ` so that `targ` can be referenced as your final switchable item.

If, in the current iteration of the loop, you have not found a valid item, the first thing to do is check whether the `vlen` variable is less than the current value of `dist`. `dist` is used in the comparison in this case because `ValidateSigilsInMap` modifies the value, further refining the distance that can be checked for valid entities. If the distance is valid for the entity, the entity is checked to see if it is either

- A health pack
- Armor
- A weapon (assuming no preferred item was found previously)

If any of those checks are true, then a valid item has been detected. `foundItem` is flagged as true, the `dist` value is updated in the same manner as before, and the `targ` variable is assigned the current entity. A final check on the item is performed to see whether it was a health pack or armor, and if so, `foundItemPreferred` is flagged as true.

The last part of the function checks the `foundItem` value, and if it is true, the final assignments to the `targ` entity occur. This changes the item into a sigil, and finishes by removing the entity that actually performs the conversion process:

```
if (foundItem)
{
    item = BG_FindItemForPowerup(PW_SIGILWHITE);
    targ->s.modelindex = item - bg_itemlist;
    targ->classname = item->classname;
    targ->item = item;
    targ->s.powerups = PW_SIGILWHITE;
    teamgame.sigil[2].entity = targ;
}

// kill the entity that does the spawn conversions
G_FreeEntity(ent);
}
```

Note that in the preceding snippet, you assign the new converted entity to `teamgame.sigil[2].entity`; this will be important later on as you start communicating the status of the sigils to the `cgame` code.

The last upkeep you need to perform is prototyping this function, because it is called from `g_spawn.c`. Open `g_team.h` and, at the end of the file (after the declaration of `Sigil_Touch`), add the following line:

```
void ValidateSigilsInMap( gentity_t *ent ); // dtf
```

Perfect! All the code is now in place for the new *DTF* game type! Go ahead and build your `cgamex86.dll` and `qagamex86.dll`, drop them in your `MyMod` folder, and load *Q3* with a `g_gametype` of 5. Launch a *CTF* map, such as `q3ctf1`. You might also want to bump the `capturelimit` of the game up to something a little more reasonable, like 100. A command-line shortcut for this would be as follows:

```
quake3.exe +set fs_game MyMod +set sv_pure 0 +set g_gametype 5 +set capturelimit 100 +map q3ctf1
```



You should see your new *Defend The Flag* described in the loading window; when the game begins, go ahead and add yourself (and a few bots) to different teams. Try picking up the flags; you should see your new sigil behavior in action when the flags aren't picked up, but instead change color. You should also notice the score slowly incrementing over time as you touch flags.

## Summary

In this chapter, you jumped through some hoops necessary to build a new game type into *Q3*. You defined a new item entity type, the sigil, and learned how to reuse existing item data by borrowing the initialization requirements for the *CTF* flags. You also learned how to create new behavior for the sigil items, allowing them to change color when touched and increment team scores through the use of a `think` function. You also got a taste for tricking *Q3* into reusing old spawn points for your new sigils, either by statically replacing specific items like the *CTF* flags, or by dynamically replacing an item using its location in the map as a guide.



# **CHAPTER 8**

# **WHERE TO GO NEXT**

**W**ell, you have finally reached the end. You've come a long way from that first rocket-launcher modification, and it is my hope that during the process you've uncovered some exciting elements in modding the *Q3* code base. My goal was to present you with the important basics of *Q3* mod development, and throughout these chapters you were shown a wide variety of fundamentals that are integral to any *Q3* engine-based game. By working directly with the game and `cgame`, you should now have a much firmer grasp on the client/server concepts that are implemented in *Q3*. That experience will carry over into many other forms of game development, because online gaming is definitely a trend that won't be going away soon.

At this point, you should have enough familiarity with the *Q3* code base to continue exploring its intricacies in greater detail. There were, however, many areas that, due to the size of this book, couldn't be covered in great detail, such as working with bots. I think that as you extend your understanding of the code base from now on, you should have a much easier time understanding what is going on; you have the tools necessary to begin educating yourself.

Open-sourcing game logic is becoming more important to developers as of late, and I encourage you to download and sift through as much source code as you can get your hands on (as of this writing, the source to *Return to Castle Wolfenstein* and *Soldier of Fortune II*, two other *Q3* engine-based games, have been made public). It also won't hurt to further your knowledge of C and C++, and get better acquainted with physics and trigonometry. In this next section, I'll give you a summary of what kinds of mods are being developed today, by fans that hunger for development like you. I'll also give you a break-down of other components that are important to mod development, outside of the C source.

# Deathmatch, CTF, and Other Game Types

Some programmers these days believe that nothing is new, and that each game that hits the shelf is simply a remake or an update of an idea that someone has already used. Whether or not you agree with this statement is your own business. Although I certainly see a trend of games falling into particular genres, what excites me about game evolution is the refinement that takes place in the process. In some cases, it is perfectly acceptable to leave well enough alone; if a certain style of game works, there is no sense in changing it. However, there are still many games that leave room for improvement, and this analogy can be applied directly to the style of game types that are offered in *Q3*, namely deathmatch and *Capture the Flag (CTF)*. Mod developers continually build on what currently exists, changing rules, updating weapons, and so on, and many of these experiments in coding alchemy often result in gold. Let's take a moment to take a look at some of the more popular game types played in FPS games, and how they have been refined over time.

## Vanilla Deathmatch

In a *deathmatch* mod, players are pitted against one another in a free-for-all battle to the death. This type of mod focuses on the skill of the individual, and how he ranks against others in the arena. Deathmatch mods almost always give the individual a full arsenal of weaponry, and impose no limits on how they wield these weapons in battle. Frag counts (the total number of kills made) are generally high, and quick to grow, and the winner is the one who comes out with the most frags by the end of the match. In most deathmatch mods, the end of the match is triggered by either a fixed time limit or by a certain score.

Deathmatch mods appeal to many gamers because of their focus on individual ability. Some players feel that pure deathmatch allows them to perfect their gaming skills, leaving them free to focus on killing every other player on the map. To them, strategy comes from determining where their opponent will hide or surprise them with the next attack. Deathmatch is also about hand-eye coordination, so that every

movement and aim is perfectly executed (pardon the pun). Most deathmatch mods truly shine when individuals are pitted one-on-one, such as in the *Rocket Arena Q3A* mod. The game becomes quite an adrenaline rush when you are constantly trying to outsmart and outshoot another player.

Deathmatch mods can definitely be built upon simply by changing the interaction of the player to the game. An example of this is the queue that is implemented in *Rocket Arena Q3A*; in this game, two players battle while remaining users wait their turn. Other mods, such as *Rune Quake*, change deathmatch by offering additional powerups and weapons to the player during the game. These simple additions may give the player an additional edge that he might not have otherwise, and can sometimes mean the difference between a second-place finish and a first-place win.

When developing deathmatch-style mods, there are a number of factors to consider. First and foremost: Are the rules you implement going to detract from the fast pace of the game? I'm sure the developers of *Rocket Arena* initially struggled with the idea of the queue because it ultimately forced players to wait, doing nothing but watching. After all, newcomers to a *Rocket Arena* server might not be up to speed with how the mod works; when they realize they must wait in line to play, they may decide to leave. Obviously, turning away players is not the first thing a mod author strives to do.

Another issue is balance. When adding a new powerup to the game, a few things need to be considered. Will this powerup grant the player too much strength? Will it make him too weak? The last thing any mod programmer wants to see is one player running around in his new mod, annihilating every other player, simply because he discovered that one super-duper powerup that granted him unlimited power. As any developer you talk to will attest, balance is key. In the end, you will never be able to satisfy *all* players, but if you do your best to please the majority, you will have succeeded in your implementation.

## Games Without Frontiers: CTF

The second type of game play that has gained popularity in recent years is the *team-based* mod. Various team-based mods began popping up here and there after the release of the initial *Quake* back in July

of 1996, but none have had a greater impact on the genre than the creation of *Capture the Flag* by Dave “Zoid” Kirsch. In his own words, Kirsch felt that existing team-based play was weak, and consisted only of “not shooting players the same color as you.” *CTF*, on the other hand, required a little thing called teamwork.

At its core, the game was still *Quake*, and players were still able to blow each other to smithereens. The success or failure of the team, however, was not based on the number of frags accrued, but instead, by the number of *captures* a team had successfully completed. To capture a flag, players needed to race into their opponents’ territory, grab the flag from its post, and sprint home—trying desperately not to be killed.

If a player brought an enemy flag to his home flag, that player would score a capture. A death, on the other hand, resulted in the flag being dropped; the flag could then be picked up by another team member or sent back home by the enemy’s touch. Thus, a new scoring system had to be implemented, one that rewarded the player for capturing the enemy’s flag as well as acknowledging when a player successfully defended his own flag from capture or returning the flag if it had been stolen.

In his initial release of *Threewave CTF* in October 1996, Kirsch implemented existing models of keys from *Quake* as the flags, and he converted four *sigils*, also pre-existing models in *Quake*, into runes that would grant the user speed, power, defense, or regeneration when picked up. Swarms of players began flocking to new *CTF* servers, and it proved to be quite an addictive mod. The popularity of *CTF* grew to such extremes that Kirsch himself ended working for ID, implementing *CTF* into both *Quake II* and *Q3*.

When designing a mod like *CTF*, the rules of the game are relatively easy to implement; the difficulty lies in the balance of the levels, and how they are designed. Some *CTF* levels allow for multiple entrances to each *flag room* (where the team’s flag is hidden); quite often, too

## NOTE

Coincidentally, Kirsch not only programmed the code for *CTF*, but also designed several of the first *CTF* levels. As a result, he had the benefit of being able to share his design vision by example instead of by direction.

many (or too few) paths to the flag room can make a *CTF* level virtually unplayable. This, of course, is in the hands of the level designer, and as the mod author, it is your responsibility to nip these issues in the bud before your mod is released to the public.

## **A Class Act: *Team Fortress***

Another popular team-based mod is *Team Fortress*. Led by developer Robin Walker, a group of *Quake* fanatics decided to build upon the success of *CTF* by pushing it one step further and implementing player *classes*. That way, not only did players have to work together to defeat their opponents, they were granted different abilities in the process. These new abilities ranged from being able to heal fellow teammates with a medical kit, to the ability to disguise themselves as members of the opposing team.

This is an exciting style of mod to create; as you can imagine, there are even more items to think about when designing it. Not only do the *CTF* level-design issues apply, but another problem arises: Are the classes properly balanced? (See, I told you balance would come up again.) By breaking out the players into multiple classes, each with its own specific strengths and weakness, imbalance may occur if you don't pay close attention to how they are used. For example, a player class that is granted heavy weaponry capable of doing twice (or more) the damage as a normal player is clearly going to be more powerful than any other class. The quickest solution to this is to slow the player class's movement. If a player can't walk as quickly, he will have to work extra hard to stay alive so he can use that powerful weaponry.

And what about the case of the spy class mentioned earlier, in which a player is capable of resembling the enemy? There is plenty of room for imbalance there. In *TF*, the solution was to lighten the class's weaponry somewhat. In addition, if the player were to fire his weapon at any time while in disguise, he would instantly blow his cover. This forces someone playing the spy class to play smart, and not just blatantly put on a disguise and trot over to the enemy's bunker. (I'll admit it looks pretty fishy when one of my own team members magically appears in the enemy's bunker and starts running toward my base.)

**TIP**

Play testing is the key to discovering balance issues, and often allows you to see things from a different point of view. When id Software was developing the mechanics behind the rocket launcher in the original *Quake*, it had absolutely no idea that players would start using it for rocket jumping. Rocket jumping is the act of pointing the rocket launcher at the ground, firing, and then jumping a split-second later, blasting the user up into the air. This technique allowed many experienced players to get themselves into very high areas of levels no player could reach by normal means. In response, id began to use rocket jumping to test new level designs, and continues to do so.

## Squad Tactics: *Counter-Strike*

*Urban Terror* may be the current hot mod for *Q3*, but it wouldn't even exist if it weren't for the path laid down by one of the most original *squad-based* mods around. That mod is *Counter-Strike*, and it was created by Minh "Gooseman" Le for the *Half-Life* engine. Unlike *CTF* or *TF*, in which there is a common goal of capturing a flag, *CS* gives each team a unique goal. As well, in *CTF* and *TF*, play is continuous; when you die, you re-spawn and get right back into the fray. In *squad-based* mods like *CS*, however, you sit out for the remainder of the round when you die. This forces a whole new set of priorities onto the player; success is earned by strategizing, using military-style tactics—a far cry from blasting a roomful of aliens with a plasma rifle. Also, unlike *TF*, *CS* does not implement a class system; instead, it allows players to purchase guns and ammo based on their previous successes, which earn them money. In my opinion, *CS* is probably one of the most realistic implementations of a mod to date, because it focuses on player movement, locational damage, and even uses weapons modeled closely after real-life counterparts.

In *CS*, two teams are pitted against each other—the evil terrorists and the heroic counter-terrorists—in a variety of missions. In one



scenario, a player on the terrorist team is given a bomb, and his teammates must help escort him to a bombsite and plant the bomb. The counter-terrorists, of course, are assigned the task of preventing this bomb from being planted. If the bomb is set, it is up to the counter-terrorists to disarm the bomb. In another scenario, the terrorists hold hostages in a specific room in a map. The counter-terrorists must charge in and rescue the hostages by any means necessary—be it offing the terrorists or safely escorting the hostages to a safe point in the map.

*CS* is an extremely popular squad-based mod, but it wasn't always so. Just as there are issues in *CTF* and *TF*, so too do problems plague mods like *CS*. In fact, the first releases of *CS* were very unbalanced, due to the weapon-purchasing system mentioned earlier. The initial implementation was “Winners get money; losers get nothing.” Although that sounds perfectly valid in theory, game play is a different story. Players soon found that after losing only a few rounds to an opposing team, the game was essentially over; the winners had amassed so much money and heavy weaponry that the losers simply could not catch up. (Keep in mind that every time a player died, he would lose all his money and additional weaponry!) A losing team armed with nothing but pistols, no matter how skillful, could not compete against a team stocked with sniper rifles, machines guns, and smoke grenades.

The eventual solution was that more events cause money to be earned, even if it is simply placing a successful shot on an opponent. In addition, even when a team loses a round, it still earns a small amount of money. This kept the winners strong, but did not penalize the losing team for long, because they could eventually purchase weapons that would get them back into the game for keeps. Earned money was also distributed among the team, so players could help each other out.

As you can see, there are certainly a lot more issues to consider when developing a squad-based mod.

## NOTE

Robin Walker's *Team Fortress* mod, along with Minh Le's *Counter-Strike* mod, gained such popularity that both mods were snatched up by Valve Software in order to implement future versions into its next generation of game technology.

However, if your goal is to encourage teamwork, strategy, and offer some inventive ways of handling weapons, skills, and powerups, it's definitely a game type to tackle.

## Structure of a Mod

It's time to take a look at the physical structure of mod. This is the breakdown of all its parts—a detailed analysis of each type of file and the role it will play in the mod. Although you have been dealing directly with the brains behind the mod (the game logic wrapped neatly into DLLs), there are many external parts, each playing an important role. In order to see this structure, I will demystify that sacred and holy ground that is known as the PK3 archive file.

### The PK3 File: Unlocking Its Secrets

*PK3* is simply a file format that id Software decided to use to store all the external data necessary to load and run *Q3*. Not only does the PK3 file format impose a bit of organization to the massive amount of files involved, it also provides an additional layer of security to *Q3*: If a server is running in pure mode, client data files must match the server's data files, forcing only those data files inside the PK3 archives to be used.

In reality, a PK3 file is simply a standard ZIP file, which can be handled by any popular zipping program. My personal choice is WinZip. If you look in the `/quake3/baseq3/` directory, you should see a list of files with the PK3 extension. With the current 1.31 Point Release installed, I see `pak0.pk3` all the way up to `pak7.pk3`. As you can see, the first one, `pak0.pk3`, is a whopper of a file, ringing in at a hefty 457MB (479,493,658 bytes!). To crack open this file, simply rename the extension so that the file is named `pak0.zip`, and then open it with a file-compression utility.

#### TIP

Some configurations of Win32 operating systems have built-in support for ZIP archives, so you may be able to view the file simply by double-clicking it in Windows Explorer.

On slower systems, it may take a few minutes to open an archive file this size, so don't worry if it is slow to view. Once the archive is visible, however, you will immediately see that there are many subfolders hidden within the file. If you have the option to sort by folder or path, do so (typically, you simply click on the column named "Path"). This way, you can have a better view of the file-tree hierarchy as it sits in the PK3 file.

Because the PK3 file rests in a subfolder of the *Q3* installation (remember, you got this from `/quake3/baseq3/`), all subfolders found within the PK3 are relative to that main subfolder. You may recall this when you dealt with shaders, icons, and sound files in Chapter 6. Table 8.1 shows the tree hierarchy and what important files are found there.

**Table 8.1 The pak0.pk3 Folder Structure**

Folder	Files Found
(root)	This directory contains configuration files for various players' key bindings and server setup.
botfiles\	This directory contains base script files that are parsed by bots to interact with players.
demos\	This directory contains demo files used to play back sessions of previously recorded games.
env\	This directory contains images used as environment maps—the areas of sky that are visible when the current level has no roof.
gfx\	This directory contains images used for fonts, blood spurts, smoke trails, crosshairs, and such.
icons\	This directory contains images used to represent items on the HUD, such as the selected weapon, ammo, flag status, and so on.
levelshots\	This directory contains screenshots of levels that are used during the pre-load stage, so the player gets a visual of where they will be fighting.
maps\	This directory contains the actual map files that are loaded and played on by <i>Q3</i> players.
menu\	This directory holds all the image art used to present the user interface to the player. Here, you will find buttons, tabs, and arrows, as well as medals and icons used in the scoreboard.

Undoubtedly, *Q3* contains a lot of important data outside the code. By peeking into this PK3 file, you'll get the idea that it can really be an intimidating task to develop a game, because so much content needs to be provided. Now that you have discovered what files are actually used in the production of *Q3*, let's spend a bit of time on the most important types of files found and the tools involved in their creation; those being artwork, models, levels, and sound effects.

Folder	Files Found
models\	This directory includes images and 3D model files that are used to represent every polygonal figure you see in <i>Q3</i> . Every player, ammo box, weapon, flying gib, and powerup will be found in this folder.
music\	This directory holds the music you hear while you play <i>Q3</i> .
scripts\	This directory holds the shader scripts.
sound\	This directory contains all the sound effects you hear, such as railgun shots, explosions, player screams, and the booming voice of the announcer.
sprites\	This directory contains the images that are used as animations to represent things like plasma blasts, bubbles in water, and the "talk" cloud that hovers above a player's head as he types a message.
textures\	This folder holds all the textures that are applied to the surfaces of maps, re-creating the dark, metallic-gothic feel of <i>Q3</i> . Animated textures are also found in this directory, such as bolts of electricity that surround some pillars in various <i>Q3</i> levels.
video\	This folder contains the in-game cinematic movies you get to see as you progress through <i>Q3</i> .
vm\	This directory contains the <i>Quake</i> virtual machine files for the three modules: game, cgame, and ui.

## Art Is Life (and Death, in *Q3*)

No matter where you look in *Q3*, you are ultimately looking at an artist's view of the world. The artist is responsible for drawing every texture, every explosion, and every icon you see on your HUD. That means in order to come up with the art for *Q3*, you must have a number of tools at your disposal (as well as an actual talent for drawing). The first of these tools is an image-editing application. This can be anything from a shareware program such as Paint Shop Pro, all the way up to a full-scale professional graphic-design application such as Adobe Photoshop.

Typically, if you are creating art for *Q3*, you will be working with two types of images. The first of these image types is the *JPEG* (Joint Photographic Experts Group) format. This type of image offers you a good range of compression (which means that the final image file is fairly small) without too much of a loss of quality. The second type of image format used in *Q3* is the *Targa* file, or *TGA* for short. Targa files are among the most frequently used image formats today because they can store high-quality image data and support such features as the *alpha channel*, which allows for transparency in *Q3*. Figure 8.1 shows a TGA file, the Capture medal from *Q3*, being edited in Adobe Photoshop.

Another tool used by artists is a piece of hardware called a *scanner*, which allows you to convert a hand-drawn image on a piece of paper or a photo into an image on a computer screen (this is known as *digitizing*). This removes a lot of the grunt work often associated with drawing art directly in a graphics-editing program. (If you've ever tried to sketch something in a painting program with your mouse, I'm sure you can relate!) Although many professional designers have become adept at using the mouse while drawing, a scanner is definitely a good idea for budding and enthusiastic newcomers.

## Modeling Without a Runway

Because the world of *Q3* is rendered in a real 3D environment, objects must have volume in order to look realistic. No, I don't mean they have to be loud; by *volume*, I mean they must have a physical height, width, and depth. In the days of *DooM*, all characters were rendered on-screen as two-dimensional images called *sprites*, which looked very



**Figure 8.1** Editing a Targa file in Photoshop

similar to a figure on piece of paper held up in front of your face. The pieces of paper . . . er, sprites in *DooM* were coded to always face the player, no matter what direction the player. It was a neat trick, but not very realistic.

Enter the 3D world of *Quake*, in which characters are represented by polygons. These polygons are placed together, like pieces of a puzzle, to form basic three-dimensional shapes, or *primitives*. Some examples of primitives are cubes, spheres, cones, and cylinders. Hundreds (and sometimes thousands) of these primitives are then fitted together to represent identifiable figures, called *models* that have height, width, and depth. In addition to representing players, models can also depict any other type of object in a game, be it a flag, monster, gun, or anything else you can imagine. These models can be rotated, scaled (made smaller or larger), and skinned (have images applied to the outside surfaces of the model). Models in *Q3* are created through the use of a number of utilities, the most popular of which is Milkshape 3D. With this tool, you can create a three-dimensional figure that can be imported into *Q3*.

In order to simulate real-life objects, models are animated in the very same manner as cartoons. Model animation is achieved in a multitude

of ways in today's games; for *Q3*, animation occurs through the creation of multiple *frames* of the model in action. When this model is placed in the game, *Q3* then begins to calculate all the missing information between each frame of animation. This causes the model to distort or *morph* between frames, resulting in a smooth transition. Figure 8.2 demonstrates the creation of a flag model in Milkshape 3D.

## Brushing Up on Level Design

If the models in *Q3* are rendered in true 3D, you should expect that the world itself exists in a 3D environment as well. Walls, roofs, arches, trap doors, tubes, and more are all perfectly rendered, thanks to an ingenious design that has been perfected since the original *Quake*. The world of *Q3*, amazing as it is, is built upon a fundamental coding theory, called *BSP tree*, which has existed since at least as far back as 1980. BSP (Binary Space Partitioning) tree is the format used by *Q3* to represent the world to the player. Thankfully, the complicated definition of a BSP tree is not one that even a level designer needs to worry about, because there are tools to assist in the development of these levels. One specific tool is *Q3Radiant*.

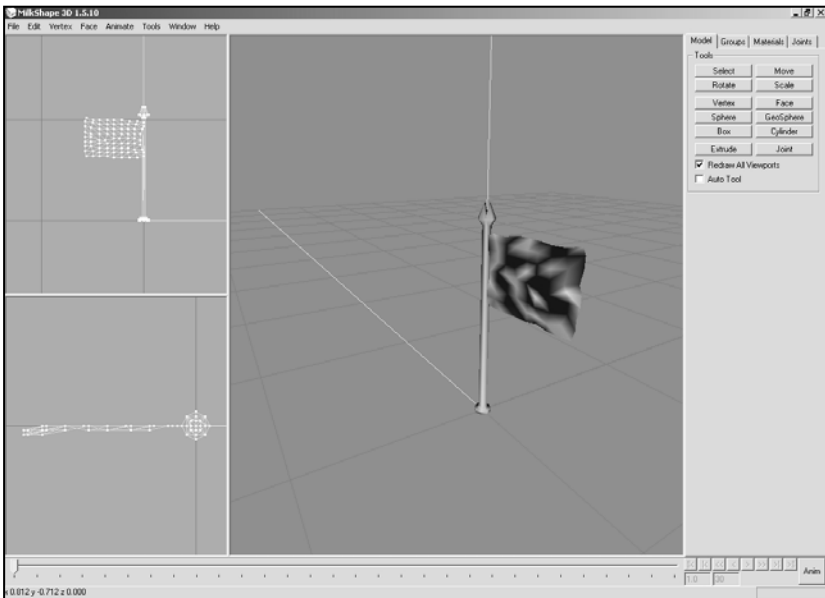


Figure 8.2 A flag model loaded into Milkshape 3D

Q3Radiant was created by id Software to quickly develop levels that players move through within *Q3*. The concept of a level is very simple: You start with basic rectangular building blocks called *brushes*, and place them next to each other to construct the foundation of a level. These brushes can be further honed and tweaked to the designer's liking; for example, holes can be punched through them, and curved surfaces (also referred to as *patches*) can be created. As well, brushes can have certain properties assigned to them, allowing them to move, like doors or elevators. Note, too, that brushes need not be solid; a pool of boiling lava is also a brush.

A brush can even be invisible, such as a transparent wall placed in front of a door, so that when the player moves toward the door and touches the invisible brush, it triggers an event, like lights in that room. These dynamic brushes that can move or trigger events are known as *brush entities*. Figure 8.3 shows a level being built from the ground up in Q3Radiant.

Brushes are not the only makeup of a map; levels also contain entities with which you, the programmer, are probably more familiar. Spots to spawn players, ammo, and flags are all examples of entities that must be positioned in a map. As well, lights must be placed throughout the map, with both their color and intensity specified, to give a room just the right mood. Level design not only takes the steadfast eye of an architect, but a strong familiarity with Q3Radiant. It can be quite intimidating to come up with level designs that are both detailed and lifelike, while at the same time fun to play in *Q3*.

## NOTE

Level design is not without its share of troubles. One thing a level designer must do when creating maps for *Q3* is to make absolutely certain the level is sealed tight. Even the maps that seem to be open, allowing the player to see the sky, are still sealed. (Even the sky is a brush!) If for any reason a particular wall or door isn't perfectly flush with every other adjacent brush, a *leak* can occur. Q3Radiant can help designers find leaks and eliminate them, but back in the days of *Quake*, leaks had to be found by hand, giving many a map-maker late-night bouts of insomnia and vertigo.



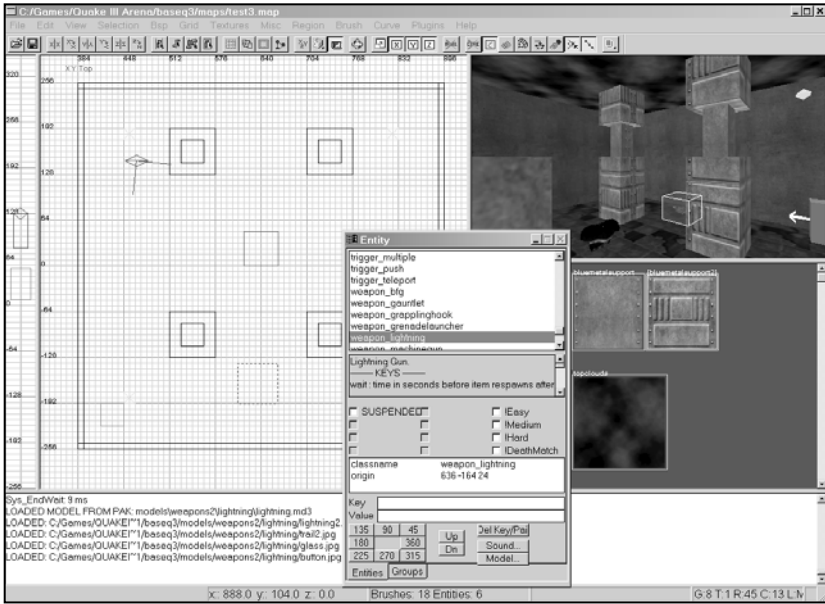


Figure 8.3 A Q3 map being constructed in Q3Radiant

## Creating Sound Effects Using Household Items

Sound plays an equally important role in the structure of a mod. Any *Q3* player will tell you that having a set of headphones on while playing *Q3* is vital; sound cues constantly tell the player what is happening to the left, right, and behind him. Because effects are dynamically mixed and played in stereo, they can pass from one speaker to the other, or be played into both ears at the same time. They can grow in intensity, and dissipate into nothingness.

All sound files in *Q3* are created using a certain resolution and sample rate. *Resolution* refers to the number of bits used to represent a sample of sound data. Typically, there are two choices: 8-bit and 16-bit. Most *Q3* sound files use 16-bit resolution, which means there are 2 bytes of data per sample (remember, 8 bits in a byte!). Obviously, doubling the data sampled (an 8-bit resolution would yield a 1-byte sample) means that the size of the sound file will be bigger, but the quality of the sound will also be twice as clear.

The second property of a sound file, the *sample rate*, refers to how many times per second a sample of data is acquired. This sample rate is measured in Hz, or *hertz*. For *Q3*, the sample rate is 22kHz, or 22,050Hz. Comparatively, CD audio tracks are typically recorded at 44kHz, or 44,100Hz. Additionally, a sound file may have multiple *channels* in which the data resides. This creates the effect of stereo: one continuous piece of sound that plays differently in each ear. Because *Q3* is constantly mixing sound effects together and playing them throughout the game, the sound data is recorded in *mono*, meaning there is only one channel of sound. This saves file size, and makes more sense, because *Q3* will play the sound in the proper speaker when the time comes.

Sound effects can be created with any kind of sound-editing utility, such as Cool Edit 2000, shown in Figure 8.4. In addition, you will probably want a microphone that will allow you to record your voice or hand-generated sound effects. Tools like Cool Edit 2000 allow you to modify the sound after it has been recorded, enabling you to achieve some



**Figure 8.4** Looking at a sample of sound data from *Q3* in Cool Edit 2000

pretty crazy effects. It just so happens that I used Cool Edit 2000 to create the sound effect of the triple award you added in Chapter 6.

As shown in this section, a lot of external data accompanies *Q3* when it loads and runs. All these assets are vital to *Q3*, and equally vital to mod development. The more detail you can add, the more control you will ultimately be able to leverage across *Q3* as you stage a development effort. In Appendix B on the CD-ROM, I'll provide tons of resources for you to further your research into one or all of the categories listed in this section. In the end, it is up to you to decide what you want to change in your mod.

## Summary

In this chapter, you looked at a gamer's perspective of game types available for play in common FPS games like *Q3*. The wide array of online gaming servers today is dominated by deathmatch and various team-based styles of game play. You also looked at the guts of *Q3*, the hidden secrets lurking within a PK3 archive—all the data necessary to make *Q3* run, act, look, feel, and sound like it does is stored in a PK3 file. You are now armed to the teeth with all the information you need to continue your adventure into the world of mod development. Good luck, and above all . . . have fun!

# CHAPTER 9

# U1 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 <i>thumb</i> 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.

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

### 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.*



**Table 9.2 Menu-Formatting Flags**

<b>Name</b>	<b>Description</b>
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;
```

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:

## 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."

**Table 9.3 Required Inits for `menutext_s`**

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

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

**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.



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_SPLLevelMenu` 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`.

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

**TIP**

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

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.

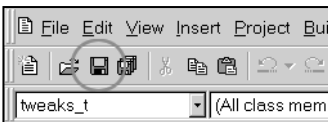
You will want to save this new text file right away so you can add it to the existing ui code (and remember, this means you will add it to the q3\_ui project). Go ahead and save the file by pressing Ctrl+S, or by clicking the Save toolbar button (the one with a disk on it), shown in Figure 9.4.

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, click 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.

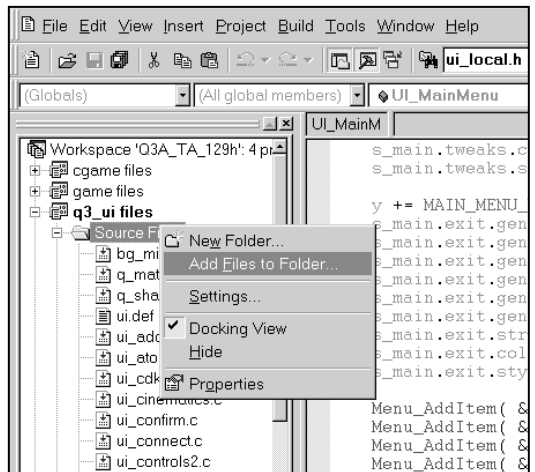
A dialog box should open, allowing you to look through folders for files to be added to the project. Find your new `ui_tweaks.c` file in the `/quake3/code/q3_ui/` folder, and add it now. When this task is



**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"
```

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

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.

**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  $320 \times 16$  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  $320 \times 130$  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
<code>generic.type</code>	This member is set to <code>MTYPE_RADIOBUTTON</code> .
<code>generic.x</code>	This member sets the control’s <code>x</code> location on the screen.
<code>generic.y</code>	This member sets the control’s <code>y</code> location on the screen.
<code>generic.name</code>	This member holds the text display to the left of the button.
<code>curvalue</code>	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
<code>generic.type</code>	This member is set to <code>MTYPE_BITMAP</code> .
<code>generic.x</code>	This member sets the control's x location on the screen.
<code>generic.y</code>	This member sets the control's y location on the screen.
<code>generic.flags</code>	If the bitmap is static and non-interactive, this member is set to <code>QMF_INACTIVE</code> ; otherwise, standard formatting flags can be applied.
<code>generic.name</code>	This member is assigned to the image filename and path to load the image. Setting this will automatically handle setting shader as well.
<code>shader</code>	This member is assigned to the filename and path of the shader, if needed.
<code>errorpic</code>	This member is assigned to the image filename and path to load if the main image in <code>generic.name</code> cannot be found or loaded.
<code>focuspic</code>	This member is assigned to the image filename and path to load if the control is active by the keyboard or mouse pointer.
<code>focusshader</code>	This member is assigned to the filename and path of the shader to be used when the control is active, if needed.
<code>focuscolor</code>	This member specifies the color of the image when it is made active.
<code>width</code>	This member specifies the width of the image.
<code>height</code>	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.frame1.generic.type           = MTYPE_BITMAP;
s_tweaks.frame1.generic.name          = ART_FRAME1;
s_tweaks.frame1.generic.flags        = QMF_INACTIVE;
s_tweaks.frame1.generic.x            = 0;
s_tweaks.frame1.generic.y            = 78;
s_tweaks.frame1.width                 = 256;
s_tweaks.frame1.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.frame1 );
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_FRAME1 );
    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 menu



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         frame1;
    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  $320 \times 150$ , 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 `menu-common_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 `TweaksSettings_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.frame1 );
    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 `menuserial_s` Control: Great for Parties

Without a doubt, the `menuserial_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 `menuserial_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 `menuserlider_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;
} menuserlider_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 `menuserlider_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 `menuser_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 `menuser_s`. An excerpt from `tweaks_t` should read like this:

```
menuradiobutton_s      thirdPerson;
menufield_s            sex;
menuser_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 `menuser_s` control, take a quick peek at Table 9.8, which lists the variable assignments necessary to use a `menuser_s` control.

**Table 9.8 Required Inits for `menuslider_s`**

Variable	Value
<code>generic.type</code>	This member is assigned a value of <code>MTYPE_SLIDER</code> .
<code>generic.x</code>	This member sets the control's x location on the screen.
<code>generic.y</code>	This member sets the control's y location on the screen.
<code>generic.name</code>	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.
<code>minvalue</code>	This member holds the slider's minimum value, which must be less than <code>maxvalue</code> .
<code>maxvalue</code>	This member holds the slider's maximum value.
<code>curvalue</code>	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.

## 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

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.

**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
<code>generic.type</code>	This member is set to <code>MTYPE_SPINCONTROL</code> .
<code>generic.x</code>	This member sets the control's x location on the screen.
<code>generic.y</code>	This member sets the control's y location on the screen.
<code>generic.name</code>	This member holds the text label that is drawn to the left of the control.
<code>itemnames</code>	This member holds the list of elements to be cycled through by the control.
<code>curvalue</code>	This member references the currently selected element in the list, which maps to an index in the array held by <code>itemnames</code> .
<code>numitems</code>	This member holds the total number of elements in the list. For the <code>MTYPE_SPINCONTROL</code> style of <code>menulist_s</code> , 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.

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

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

## 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!**

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:

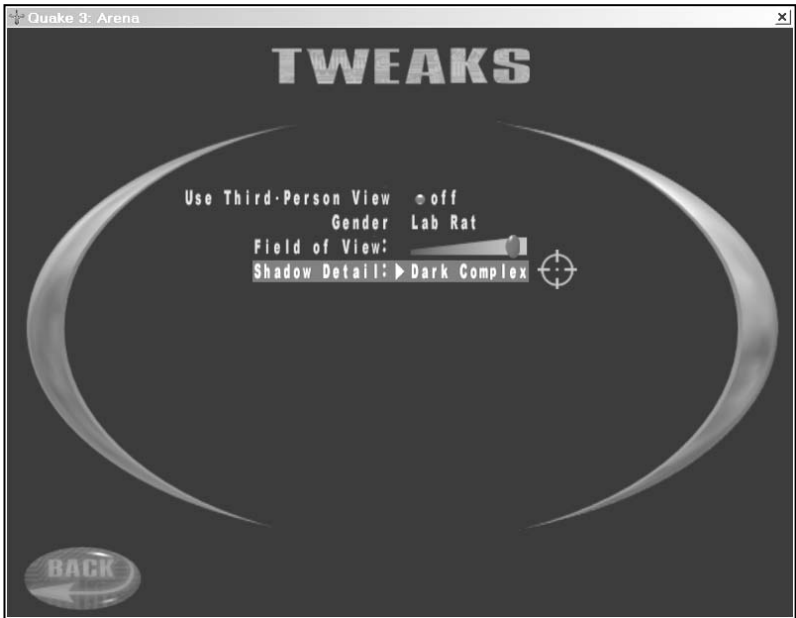
```

Menu_AddItem( &s_tweaks.menu, &s_tweaks.fov ); // new menulider_s!
Menu_AddItem( &s_tweaks.menu, &s_tweaks.shadowDetail ); // new
menulist_s!
Menu_AddItem( &s_tweaks.menu, &s_tweaks.frame1 );

```

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.

# CHAPTER 10

# ENHANCING DTF

**B**y 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 posed 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 pre-caching 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 compatibly, 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 49<sup>th</sup> 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 repre-

sensation 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.

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';
```

## 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.

```
    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  $640 \times 480$  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  $10 \times 120$ , 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  $24 \times 24$  (because the original flag status icons are  $32 \times 32$ ). 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 `cgame86.dll` and `qgame86.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





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 `cggame 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.

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 `cggame.Cvar`

### 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.**

**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 <i>read-only</i> ; 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 <code>set</code> 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 <code>cvar_restart</code> 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.persistent[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 check only 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  $640 \times 480$ , regardless of screen resolution), and variables `w` and `h` to hold half the screen's dimensions. In both cases, the values are  $320 \times 240$ . 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  $\text{width}^2 + \text{height}^2$  as the value. 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  $8 \times 8$  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 multi-player 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 `game_type_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 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_sigilLocator`, 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 ) );
```





## 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.

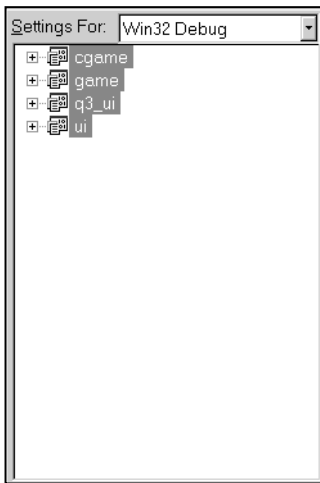


# **APPENDIX A**

# **DEBUGGING YOUR MOD IN VISUAL STUDIO**

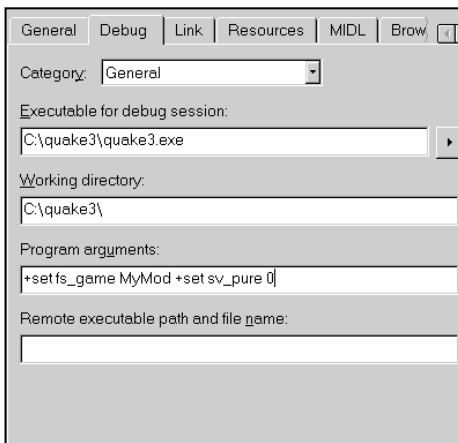
A programmer requires an extensive set of tools to get his job done; working with C can often be a frustrating and confusing process. Bugs that pop up in your code can lead to nightmarish days and endless nights as you try to decipher the simplest of problems. Working with *Q3*'s code base is no different. One of the best tools available to a programmer when working in Visual Studio is the debugger, which lets you stop your program as it runs, watch the values of variables and the flow of logic, locate the dastardly bug, and squash it. Following is a description of how to set up the debugger in Visual Studio so that you can use it for your *Q3* mod development and testing. Start by loading the *Q3* source project into Visual Studio. Then do the following:

1. Open the Project menu and choose Settings. (Alternatively, press Alt+F7.)
2. The Project Settings dialog opens, with the subprojects *game*, *cgame*, *q3\_ui*, and so on visible on the left side, and a tabbed interface on the right. On the left, near the top, select the Settings For drop-down menu and select Win32 Debug.
3. Highlight all the subprojects (*game*, *cgame*, *q3\_ui*, and *ui*) by holding down Ctrl while you click on each subproject (see Figure A.1).
4. On the right side of the Project Settings dialog box, click the Debug tab.
5. You should see a Category drop-down menu; make sure it is set to General.
6. In the Executable For . . . text box, type the full path to your *Q3* executable (or use the right-arrow button next to the text box to browse your hard drive to find it).
7. In the Working Directory text box, type the full path to your *Q3* executable, minus the name of the executable file. So if you said your executable file was at *c:\quake3\quake3*, your working directory should only be *c:\quake3\*.



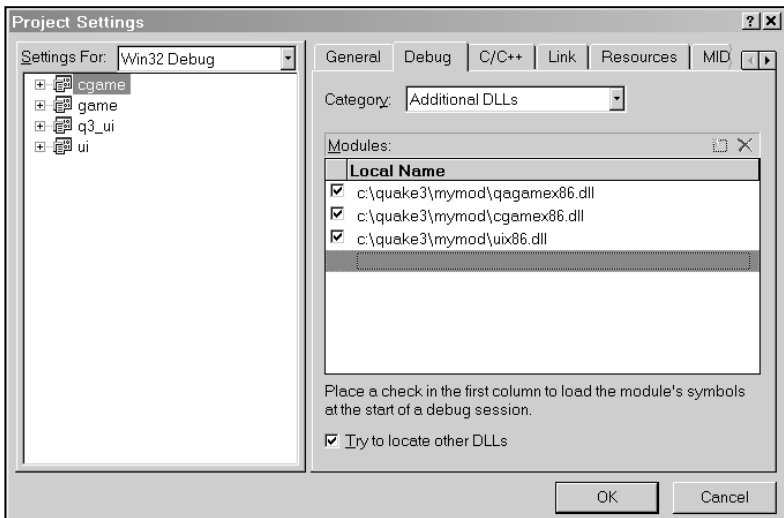
**Figure A.1** *Creating your first IDirect3D8 object*

8. In the Program Arguments text box, type `+set fs_game MyMod +set sv_pure 0`. (You should recognize these parameters as the ones you used during all the development throughout this book; if you use a different subfolder for your specific mod, make the appropriate specification instead of MyMod.) Figure A.2 shows the results of these entries.



**Figure A.1** *Creating your first IDirect3D8 object*

9. In the Debug tab, click the Category drop-down menu and choose Additional DLLs. The text boxes are replaced with a gray box labeled “Modules”; this is where you can specify any DLLs that are required during debugging.
10. You cannot add DLLs to all subprojects at once, so start by highlighting `cgame` on the left. The gray Modules box on the right should now be white, indicating that you can add DLLs.
11. Click the Add a New DLL button (the small dashed box next to the red X to the far right of the Modules label; the red X is the Remove a DLL button).
12. A new text box appears next to a new check box. Type the full path to the specific DLL you want to add (or browse your hard drive to locate it by clicking the button marked with an ellipsis). You should specify all three *Q3* DLLs in your MyMod folder—`qagamex86.dll`, `cgamex86.dll` and `uix86.dll`—as shown in Figure A.3.
13. After all three DLLs are specified, repeat the process for `game`, `q3_ui`, and `ui`.
14. Click the Link tab.
15. Make sure the Category drop-down list is set to General.



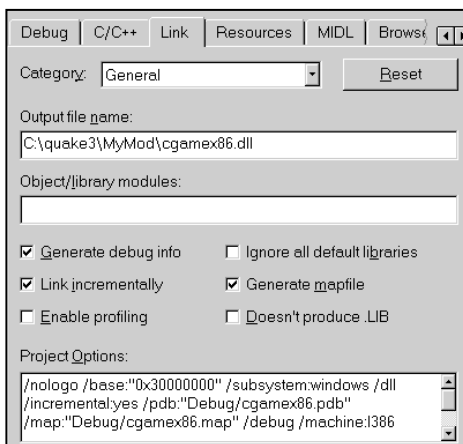
**Figure A.3** Specifying additional DLLs for a project debug session

16. In the Output File Name text box, type the full path to the final DLL that will be built when you compile.
17. Repeat step 16 for each subproject in turn, because they all build to different file names. When you're finished, you'll have specified the full path for `cgame.exe` for `cgame`, `qgame.exe` for `qgame`, and `ui.exe` for both `q3_ui` and `ui` (see Figure A.4).
18. Click the OK button in the lower-right portion of the dialog box.

**NOTE**

Make sure your output path matches your Q3 directory and your specific mod directory, which in this example is "MyMod."

To actually debug your code, you will need to set *breakpoints* in your code, and possibly use *watches* to look at current values of variables. A breakpoint refers to a specific line number within code that stops execution of the program when Visual Studio reaches it. You are able to set multiple breakpoints, if needed. Once a breakpoint is hit, the debugged program pauses, and Visual Studio takes control, highlighting the line of code that execution stopped at. A watch is used to tell Visual Studio that you want to observe the value of a specific variable



**Figure A.4** The link tab, with an output file name specified

as a program executes. You're able to drag-and-drop variables right into the watch window, as well as type them in manually.

To quickly try this out, open `g_items.c` and scroll to line 398, where you should find the opening to the function `Touch_Item`. Set a breakpoint on this function by clicking the Open Hand icon on your Visual Studio toolbar (see Figure A.5), or by simply pressing F9.

The result of this key press is that a maroon dot is placed to the left of the line of code on which you set your breakpoint. Pressing F9 again will remove the dot. With the maroon dot in place, do the following:

1. Start a new debug session by clicking the Go button (see Figure A.6) or by pressing F5.
2. Visual Studio launches *Q3*, as shown in Figure A.7. With a live debug session in progress, load a map and touch an item.
3. The *Q3* window disappears, and Visual Studio takes over, using a bright yellow arrow to point to the line in your code where the program has paused. You can now step through the code, line by line, by clicking the Step Into button (see Figure A.8) or by pressing F11.

### TIP

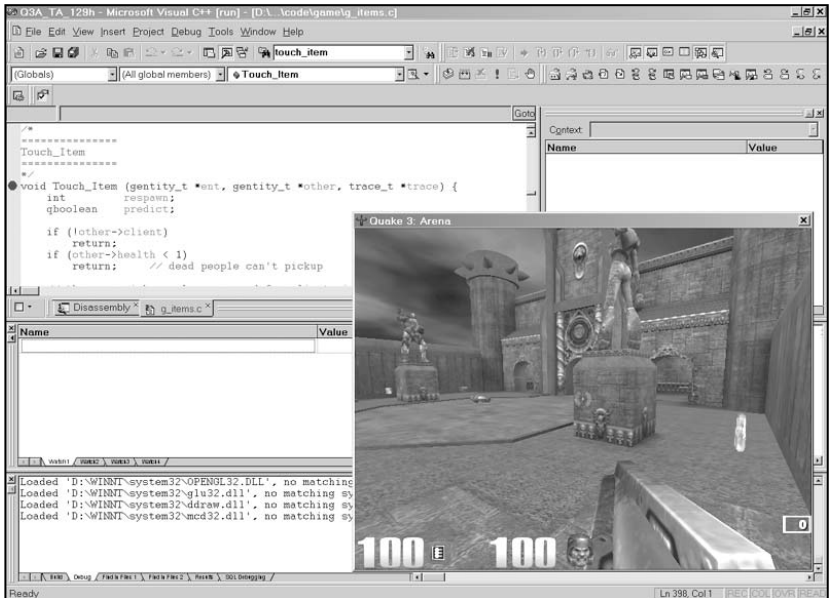
I highly recommend going to the *Q3* system settings and disabling full-screen mode so that you can debug *Q3* while it runs in a window. This will make switching back to Visual Studio extremely easy when a breakpoint is hit. You can toggle the full-screen option in *Q3*'s System menu, under Graphics.



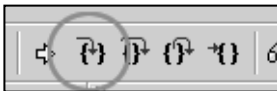
**Figure A.5** *The Breakpoint button*



**Figure A.6** *The Go button*



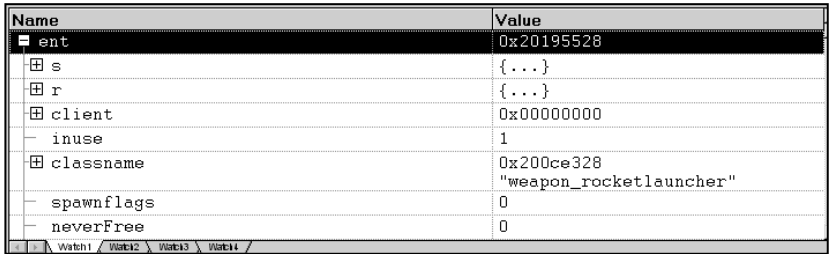
**Figure A.7** A Q3 debug session in action



**Figure A.8** The Step Into button

- To examine the specific values of variables during this pause in the execution of *Q3*, simply highlight it and drag it into the Watch debug window. (To make this window visible, press Alt+3.) As shown in Figure A.9, the Watch debug window has two columns, *Name* and *Value*, which refer to the variable and its contents. Because *Touch\_Item*'s first input parameter is *ent*, which should represent the touched item in question, select the word *ent* and drag it to the Watch window. It should populate into a tree hierarchy, which you can break out by clicking the + sign next to each complex variable.
- When you are ready to return to *Q3*, simply click the Go button or press F5, or stop debugging altogether by opening the Debug menu and choosing Stop Debugging or pressing Shift+F5.





Name	Value
ent	0x20195528
s	{...}
r	{...}
client	0x00000000
inuse	1
classname	0x200ce328 "weapon_rocketauncher"
spawnflags	0
neverFree	0

**Figure A.9** *The Watch debug window*

I encourage you to spend time getting to know the debugger and reading up on the subject, if possible. There are many more details that the debugger can manage; if wielded properly, it is a powerful tool for fixing code. A good starting point is Microsoft's MSDN website, which offers detailed information on Visual Studio's debugger, at <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsdebug/html/vchowViewingModifyingData.asp>.



# APPENDIX B

# RESOURCES

In order to guide your exploration effort, furthering your level of expertise in the art of modding, I have assembled a list of Web sites that I think will benefit you greatly. They cover a wide range, from code-related sites, to general mod development, to game-business information sites, and beyond.

**Code3Arena:** <http://www.planetquake.com/code3arena/>

This is the site where many *Q3*-mod programmers (including myself) got their start, thanks to the dedication and tenacity of its contributors. Here, you will find many tutorials on weapon upgrades, HUD improvements, game rule changes, and more. They also document many *Q3*-modding concepts, such as entities, vectors, Cvars, and the UI.

**Quake DevelS:** <http://www.planetquake.com/qdevels/>

Although this Web site focuses on programming mods for *Quake II*, many of the concepts and tutorials apply to *Q3* mod programming as well.

**Quake III Arena Shader Manual:** <http://www.heppler.com/shader/>

The fine art of working with shaders is a must for all *Q3* programmers; this Web site contains the best documentation on the Net for such purposes. Coincidentally, Paul Jaquays and Brian Hook, two of the original developers on *Q3*, wrote it.

**Milkshape 3D:** <http://www.milkshape3d.com/>

One of the most widely used tools for creating 3D models can be found here, at Chumbalum Soft's Web site. Not only can you purchase the full tool at this site, you can also find tutorials about getting started with modeling.

**Radiant:** <http://www.qeradiant.com/>

Radiant's Web site is *the* portal for level-design information and tools. Here you will find downloads, tutorials, manuals, and even forums in which to participate if you want to explore the mystery of level design for *Q3* and similar games.

**Polycount:** <http://www.planetquake.com/polycount/>

This Web site is about the coolest online resource around to satisfy your 3D-modeling curiosity. Pay Polycount a visit to get caught up on the latest news, tutorials, and tools used by budding 3D modelers. There is a pretty hefty archive of user-created models to download as well.

**GameDev.net:** <http://www.gamedev.net/>

Without a doubt, one of the largest and most extensive Web sites on the Net dedicated to game development is GameDev.net. You'll find a vast array of forums, as well as articles, news items, recommended books, and more.

**FilePlanet:** <http://www.fileplanet.com/>

FilePlanet is one of the largest online hubs for downloading game demos, movies, patches, mods, and more. Not only can you find a wide variety of professional and user-built mods at this site, FilePlanet will also help you get *your* files out to the public when the time comes for you to make your hard work available.

**Syntrillium Software:** <http://www.syntrillium.com/>

If you want to start experimenting with creating and editing audio for your mods, look no further than Cool Edit 2000. You can manipulate all kinds of audio data formats with this tool, including MP3s, and apply all kinds of sound effects and filters to recreate that perfect robotic voice or massive explosion you need.

*This page intentionally left blank*

# Index

## A

- absmax game coordinate**, 127
- absmin game coordinate**, 127
- absolute coordinates**, 130
- accuracyFactor**, 61, 67
  - for shooting while moving, 68, 70
- accuracy\_hits variable**, 101
- accuracy variable**, 55
  - adjusting, 58–62
- add function**, 34
- Adobe Photoshop**, 264
  - Targa files, editing, 265
- analog controls**, 122
- animations**, 265–266
- Apocalypse Void*, 119
- arc variable**, 168
- area of effect for gravity wells**, 88–89
- armor regeneration**, 177–179, 182–184
- arrays**, 56
  - size of function, 84
  - static array, 157
- artwork**, 264
- Asteroids*, 126
- authoritative update**, 140–141
- awards**, 184–193
  - caching, 194–195
  - calling, 196–200
  - changes in, 197–198
  - ClientThink\_Real and, 192–193

- displaying awards, 193–196
- flags for, 186–187
- logging awards, 187

## B

- bandwidth**, 65
- Batch Build dialog box**, 174
- batch files**, 29
- BBSs (bulletin board systems)**, 2
- beam weapons**, 51, 156–166
- BG\_CanItemBeGrabbed**, 229–230
- BG\_FindItemForPowerup**, 214
- bg\_itemlist**, 216
- bg\_pmove.c file**, 109
  - PM\_CheckJump in, 112–113
- binary format**, 188–189
- bit flags**
  - for awards, 186–187
  - PMF\_JETPACK flag, 115–116
  - uniqueness of, 117
- bits**, 188–189
- bitwise and assignment operator**, 63
- bitwise exclusive or assignment operator**, 63
- bitwise operators**, 62–63
- bitwise or assignment operator**, 63
- body parts**
  - hit flags for, 124–126
  - scanning, 128–130
- boolean variables**, 40, 55

**bouncing grenades**, 74–75  
**bounding boxes**, 84, 126–128  
**break keyword**, 165  
**brushes**, 267–268  
**BSP files**, 241  
**BSP tree**, 266–267  
**buckshot**, 54  
**bush entities**, 267  
**bytes**, 188–189

## C

**catching**  
     award icons, 194–195  
     sound effects, 195–196  
**catching data**, 178  
**CalculateRanks**, 236  
**Capture-and-Hold**, 202  
**Capture the Flag**, 186, 255, 256–258. *See also Defend the Flag*  
     definition for red flag, 217–218  
**Carmack, John**, 15–16, 18, 151  
**centity\_t**, 171–172  
**CG\_AddWeaponWithPowerups**, 181  
**cgame code**, 65, 143. *See also Defend the Flag*  
     assisting game code, 145  
     entities, list of, 146  
     files, 31–32  
     responsibility of, 145–147  
     for reward notification, 184–193  
     shaders, initializing, 183  
     for shooting while moving, 70, 72  
     sound effects, playing, 237–238  
**CG\_CheckLocalSounds**, 196  
**CG\_EntityEvent**, 170–174  
**CG\_LightningArc**, 168  
**cgMedia\_t**, 179  
**CG\_PlayerSprites**, 197–198  
**CG\_RegisterGraphics**, 178  
**cgs**, 178–179  
     meaning of, 194  
**CG\_ShotgunPattern function**, 63–66  
**cg.snap property**, 66–67  
**CG\_TransitionEntity**, 172–173  
**cg variable**, 66–67  
**ChainLightning\_Fire**, 160  
     for regenerating armor, 182–184  
**chain-lightning gun**, 156–166  
     armor regeneration and, 177–179, 182–184  
     content mask for, 163  
     firing, 165–166  
     path to target, checking, 162–165  
     reward notification, 184–193  
     tracking hit targets, 161  
     validating new target, 165  
     vectors for, 162  
     visual effects, creating, 174–175  
     Weapon\_LightningFire function, 157–160  
**channels of sound file**, 269  
**chars**, 4  
**circles for targets**, 88  
**C language**, 4–5, 14. *See also VC++*  
     bits and bytes in, 189  
**ClientCommand function**, 116  
**client counting variable**, 192  
**client events**, 166–175  
     centity\_t struct, 171–172  
     enumerating, 167–170  
**clientNum**, 66, 104  
**clientPersistant\_t**, 40, 100

code, 104–105

**clients**, 135

- dumb client, 139–140

**client/server applications**, 134–135

- architecture for, 135–137
- cgame code and, 145–147
- latency issues, 136–139

**clientSession\_t**, 100–101

- coding, 105

**client-side code**, 65

- cg variable, 66–67

**client spawns**, 95

**ClientThink\_Real**, 105–107

- for awards, 192–193
- jumping with, 109–112

**cluster grenades**. *See* grenades

**Cmd\_SetHoming\_f**, 41

**Cmd\_Stats\_f**, 116–117

**Cmd\_ToggleJetpack\_f**, 116–117

**collision detection**, 84, 126–128

**colors**. *See* *Defend the Flag*

**Com\_Error**, 230

**compiling projects**, 27–28

**console commands**, 39

- creating, 40–41
- for jetpack, 116

**constant variable, declaring**, 109

**content mask**, 163

**continue keyword**, 161

- in loop, 165

**controls, types of**, 122

**Cool Edit 2000**, 269

**copying source directory**, 21

**CounterStrike**, 6, 259–261

- locational damage system, 124

**CrossProduct**, 65, 66

- for shooting while moving, 72

**crouching players**, 59

**curly braces**, 217

**customShader member**, 182

**Cvars**

- g\_gametype, 206
- list of, 150
- setting, 150

## D

**damage**. *See also* locational damage

- bounding boxes and, 127–128
- to players, 124–126
- shotgun damage, 56–58

**damage\_armor variable**, 101

**deathmatch mods**, 255–256

**debug symbols**, 25

**decimal format**, 188–189

**Defend the Flag**, 202–252

- BG\_CanItemBeGrabbed, 229–230
- BG\_FindItemForPowerup, 214
- bg\_itemlist, 216
- CalculateRanks, 236
- capturelimit variable, 206
- colors
  - of flags, 212
  - use of, 226
- converting *CTF* flags to sigils, 244–245
- flagStatus\_t, 211–212
- gametype\_t, 206–208
- G\_CallSpawn, 240–242
- G\_CheckTeamItems, 225–226
- gitem\_t, 215–218
- G\_Spawn, 247
- GT\_DTF, 223



**Defend the Flag (continued)**

- initializing sigils, 220–225
- itemType\_t, 209–211
- IT\_SIGIL flag, adding, 244
- level.gentities, 239
- locations of sigils, 248
- picking up sigils, 230–231
- Pickup\_Team, 232
- powerup\_t, 213–215
- preferred item, picking, 249–252
- preparing *Q3* for, 205–206
- red and blue flags, modifying, 209–211
- registering entity states for sigils, 242–244
- RegisterItem call, 243–244
- respawn integer, 232
- reused code, 204–205
- scoring, 235–236
- sigils
  - actions of, 214
  - at beginning of team game, 224–225
  - converting *CTF* flags to, 244–245
  - gitem\_t, 215–218
  - initializing, 220–225
  - locations of, 248
  - new behavior, creating, 227–238
    - as new items, 218–220
    - picking up, 230–231
    - registering entity states for, 242–244
    - team\_CTF\_blueflag item, 218–220
    - teamgame.sigil, 221–222
    - think function/nextthink time, 234
    - third spawn point, creating, 245–249
    - transforming flags into, 215–227
    - ValidateSigilsInMap, 247–248, 250–251
    - warning about, 225–227
  - sigilStatus\_t, 212
  - Sigil\_Think, 235–236
  - Sigil\_Touch, 232–234
    - prototyping, 234–235
  - sound effects, changing, 236–238
  - spawn points, 238–252
    - launching entities into map, 239–242
    - process of spawning, 239–242
    - RegisterItem call, 243–244
    - third spawn point, creating, 245–249
  - SP\_worldspawn, 240
  - status of flags, 211–212
  - team-based styles of play, 207
  - team\_CTF\_blueflag item, 218–220
  - teamgame variable, 221
  - Team\_InitGame, 220–221, 222–223
  - Team\_SetSigilStatus, 223–224
  - third spawn point, creating, 245–249
  - Touch\_Item, 228–231
    - BG\_CanItemBeGrabbed, 229–230
    - wiring sigils for, 231–235
  - trap\_EntitiesInBox, 248–249
  - tvec, 249–250
  - ValidateSigilsInMap, 247–248, 250–251
  - warning about sigils, 225–227
- digital controls**, 122
- digitizing images**, 264

**dir floats**, 96

**Distance function**, 87

**DLLs (dynamically linkable libraries)**,  
16–18

- compiling projects and, 27–28
- security of, 151
- for shotgun, 67

**Domination**, 202–203, 220

**DooM**, 2

- peer-to-peer networks for, 135
- sprites, 264–265
- updated version of, 14

**DOS version of *Quake***, 16–18

**downloading within game**, 68

**ducked players**, 59

**dumb client**, 139–140

## E

**eFlags**, 104

- for awards, 186–187

**entities**, 29

- in cgame code, 146
- definition of, 37–38
- limit on number of, 38

**entityState\_t**, 173, 180

**entry strings**, 241

**enums**, 4

- for awards, 187
- for client events, 167–170

**error-printing function**, 230

**EV\_LIGHTNINGARC**, 165–166

**EV\_MISSILE\_HIT**, 165

**Excellent award**, 190–191

- handles, 196

**explode variable**, 89

## F

**feetAt variable**, 130

**Find in Files command**, 59

**findradius function**, 44

**fire\_grenade function**, 74, 78–82

**fire\_rocket function**, 34–35

- for homing missile function, 45–46

**first-person shooters**, 50

**flag model in Milkshape 3D**, 366

**flags**, 29, 96. *See also* bit flags

- award flags, 186–187
- bitwise operators, 62–63
- content mask flags, 163
- damage flags, 58
- grenades, bouncing, 75
- grouping flags, 62
- hit flags, 124–126
- means-of-death flags, 191–192
- PMF\_TIME\_KNOCKBACK flag, 90
- pmove flags, 59, 60
- for preferred items, 250
- trType\_t flags, 75

**flagStatus\_t**, 211–212

**flight**. *See* jetpacks

**flight powerup**, 119–123

**floats**, 4, 35

- variables and, 56

**forward/backward movement**, 71

**forward variable**, 73–74

**foundItem flag**, 250, 251

**foundPreferredItem flag**, 250

**fraglimit variable**, 149–150

**frames**, 266

**FTP (File Transfer Protocol)**, 138

**functions**, 4–5

format of, 33–34

shotgun, 53–54

## G

**g (acceleration due to gravity)**, 107

**g\_active.c**, 106

**game code**, 143

in client/server model, 144–145

consistency with, 145

**game project files**, 30–31

**gamex86.dll**, 17

**gauge of shotgun ammunition**, 54

**G\_CheckTeamItems**, 225–226

**gclient\_s**, 97–101

**gentity\_s**, 98

**gentity\_t**, 73–74

**G\_ExplodeMissile**, 76–77, 80

**g\_gametype**, 206

**giTag member**, 217

**gitem\_t**, 210, 215–218

definition of, 216

**giType member**, 217

**G\_LocDamage**, 128–130, 131

**glowing players, creating**, 179–182

**g\_missile.c**, 74

**G\_Missile Impact**, 80

**GoTo Definition Of option**, 114

**gravity**

changing player gravity, 107–108

increasing effects of, 108

PM\_StepSlideMove and, 122

**gravity wells**, 82–91

collision, testing for, 89–90

targets for, 83–89

trap\_EntitiesInBox, 83–87

**grenades**

bouncing of, 74–75

changing behavior of, 78–82

fire\_grenade function, 78–82

modifying, 72–82

splash damage, 75

think and nextthink variables, 76–78

think function, 76–78, 90–91

VectorSet function, 77–78

**GT\_DTF**, 207–208, 223

**GT\_FFA**, 206

**GT\_HARVESTER**, 207

**GT\_OBELISK**, 207

**GT\_IFCTE**, 207

**GT\_SINGLE\_PLAYER**, 206–207

**GT\_TEAM**, 207

**GT\_TOURNAMENT**, 206

**GVORTEX\_RADIUS**, 87, 91

**GVORTEX\_TIMING**, 87, 90, 91

**GVORTEX\_VELOCITY**, 87, 91

## H

**Half-Life**, 6, 259

**handles**, 193–196

in pushReward, 196–197

**hard-coded states**, 109

*Harvester*, 207

**haste rune**, 106–107

**headshots**

hit flags, assigning, 124–126

locational damage and, 130

switching values, 131–132

**health struct**, 96

**hexadecimal format**, 188–189

**Hilton, Chris**, 160

**history of *Quake***, 15–16

**hit flags**, 124–126  
switching values, 131–132

**hitscan weapons**, 50–51

**homing missile**, 33–36  
coding for, 38–47  
input, getting the, 39–41  
smoothing movement of, 47  
targeting a player, 41–47  
think function counter, adding, 48

**homing\_status variable**, 105

**hosts**. *See* servers

**HTTP (HyperTextTransfer Protocol)**, 138

## I

**icons**, 184–200  
handles, 193–196  
over head of player, 197–198  
for reward notification, 184–193

**ID Software**, 2–3  
FTP site, 15–16

**image-editing applications**, 264

**Impressive award**, 190

**infinite loop**, 160

**inflictorHeight variable**, 130

**input, damage function requiring**, 56–57

**installing**  
*Quake III*, 19–21  
VC++, 22–25

**integers**, 4, 35

**Internet**  
*Quake III* site, 19  
VC++ service packs, 25

WANs (wide area networks) and, 134

**interpolation**, 140–143

**inverse bitwise operator**, 62

**Invisibility powerup**, 108

**invisible boxes**, 84

**invisible brushes**, 267

**IT\_AMMO**, 210

**ITEM\_INDEX macro**, 234

**items**, 210  
preferred item, 249–252  
Touch\_Item, 228–231

**itemType\_t**, 209–211

**IT\_WEAPON**, 210

## J

**Jacques, Anthony**, 220

**Java**, 52–53

**jetpacks**  
adding, 115–123  
engaging, disengaging, 123  
PMF\_JETPACK flag, 115–116  
testing, 119

**joystick buttons**, 39

**JPEG format**, 264

**jumping**  
modifying velocity, 114–115  
velocity of player, 108–109

**JUMP\_VELOCITY**, 109  
modifying, 114–115

## K

**keystrokes**, 39  
typedef, saving with, 98

**Kirsch, Dave**, 238, 257

**L**

- lag.** *See* latency issue
- LANs (local area networks),** 134
- lasthurt\_loc variable,** 132
- lastkilled\_client variable,** 101
- latency issue,** 136–139. *See also* updates
  - description of, 138–139
  - interpolation, 140–143
  - low latency, 139–140
- Le, Minh,** 259
- left-hand coordinate system,** 56
- left/right movement,** 71
- leg shots**
  - hit flags, assigning, 124–126
  - locating shot for, 130
  - switching values, 131–132
- level design,** 266–267
- level.gentities,** 239
- lightning gun.** *See* chain-lightning gun
- lights, placement of,** 267
- Linux,** 14
- loading up mods,** 28–29
- locational damage,** 124–132
  - hit flags, 124–126
  - scanning body parts, 128–130
- LogAccuracyHit function,** 58
- looking up terms,** 114

**M**

- MacOS,** 14
- MaxCarnage mod,** 160
- MAX\_GENTITIES function,** 88
- means-of-death flags,** 191–192
- medalTriple handle,** 194–195
- medal\_triple.tga,** 198
- melee weapons,** 52
- memory addresses,** 5
  - for player movement, 111–112
- memory-clear of variable,** 170
- memset call,** 170, 222
- Microsoft Visual C++ Version 6,** 5–6
- Milkshape 3D,** 366
- mip-mapping,** 195
- missile weapons,** 51. *See also* homing missile
- mission pack for Quake III,** 101
- mixing sound effects,** 269
- models,** 94–95, 264–266
  - skinned models, 94–95
  - textured models, 94–95
- MOD\_LIGHTNING,** 191–192
- mods**
  - currently available mods, 6–12
  - defined, 2–3
  - reasons for creating, 3–4
  - structure of, 261–270
- morphing,** 266
- mouse controls,** 39
  - ui code and, 147–148
- movement**
  - bg\_pmove.c file, 109
  - player movement, changing, 105–115
  - shooting while moving, 68–72
- movement Dir,** 104
- MP3 player in Rocket Arena Q3A,** 8
- MSDN, installing,** 25
- MyMod.bat,** 29
- MyMod directory,** 21
  - Release folder, 28

**N**

**name char pointer**, 96  
**negative values of x, y, and z**, 56  
**networks**

computers on, 134–135  
 peer-to-peer networks, 135

**nextthink variable**, 37  
 of *Defend the Flag* sigils, 234  
 for grenades, 74, 76–78

**O**

*Obelisk*, 207  
*One Flag CTF*, 207, 211  
**ops, previous frame of player in**, 197  
**otherEntNum**, 66

**P**

**packets**, 137  
 lag affecting, 139  
 ping, 146  
 User Datagram Protocol (UDP), 138

*Painkeep Arena*, 9–10  
 gravity wells in, 83

**Paint Shop Pro**, 264

**pak0.pk3 folder structure**, 262–263

**patches**, 267

**peer-to-peer networks**, 135

**Perpendicular Vector**, 65

**PERS\_EXCELLENT\_COUNT**, 197

**persistent array**, 187, 190

**persistent client data, storing**, 40

**PERS\_TRIPLE\_COUNT**, 190, 192, 197

**Pickup\_Team**, 232

**Pickup\_Weapon**, 232

**ping**, 146

**PK3 archives**, 153–154, 261–263

pak0.pk3 folder structure, 262–263  
 shaders in, 176

**planning functions**, 33

**player\_die function**

awards and, 190  
 means-of-death flags, 191–192

**players**, 94–96. *See also* jetpacks

body parts, assigning, 124–126  
 glowing players, creating, 179–182  
 movement, changing, 105–115  
 multiple respawns of, 95  
 state of, 102–105  
 structure, 96–97  
 ui code and, 147–148

**playerState\_t**, 100

gravity member, 107–108  
 jumping member, 108–112  
 powerups member, 108  
 speed member, 106–107

**player\_t**, 96

**play testing**, 259

**PM\_Accelerate**, 122

**PM\_AirMove**, 118

**PM\_CheckJump**, 112–113, 113

**PM\_CmdScale**, 121

**PMF\_DUCKED flag**, 59, 61

**PMF\_JETPACK flag**, 115–116

PM\_AirMove, adding, 118  
 toggling, 117

**pm\_flags**, 104

**PM\_FlyMove**, 120–121

borrowing code from, 119–123

**PM\_Friction**, 121

**PMF\_TIME\_KNOCKBACK** flag, 90

**Pmove** function, 112–114

**pmove**s

flags, 59, 60

interpolation with, 142

**PmoveSingle**, 112–113

**pmove\_t**, 109, 110–111

**PM\_StepSlideMove**, 122

**pm** variable, 113

**PM\_WalkMove**, 113

**point**, 224

**pointer notation**, 97

**pointers**, 5

to structs, 97

**Point Release 1.31**, 20

**polygons**, 94

**positive values of x, y, and z**, 56

**powerups**. *See also* shaders

adding, 177

in *Defend the Flag*, 213–215

of `playerState_t`, 108

quad powerup shader, 176

**powerup\_t**, 213–215

**predictions**, 141–142

cgame and, 146–147

**predict variable**, 237

**preferred item**, 249–252

**preprocessor directives**, 101

**primitives**, 265

**ps struct**, 66

current value of player state in, 197

**pure servers**, 68

**pushReward**, 196–197

**PW\_ARMORDRAIN**, 177, 183

**PW\_FLIGHT** flag, 119–120

## Q

**qhandle\_t**, 178, 179

**q\_shared.h** file, 102

**q3asm**, 151

**Q3Radiant**, 266–268

**quad shaders**, 176

glowing players, creating, 179–182

**Quake**. *See also* *Quake Virtual Machine (QVM)*

DOS version, 16

install directory, 16

Win32 version, 16

**Quake III**

pure servers, 68

rocket launcher, 26–27

**QuakeC** language, 15–16

**QuakeCon**, 238

**Quake III Fortress**, 8–9

**Quake Virtual Machine (QVM)**, 18,  
150–154

building QVM, 152–153

communication through, 151

deploying, 21

private memory space, 151

**queues**, 8

## R

**realism**, 124

**refEntity\_t**, 168–169, 180

**RegisterItem** call, 243–244

**resolution of sound**, 268

**respawn integer**, 232

*Return to Castle Wolfenstein*, 51, 52,  
254

**reward notification**. *See* awards

**REWARD\_SPRITE\_TIME**, 192

*Rocket Arena*, 7–8, 256

rocket jumping, 259

**RT\_LIGHTNING**, 170

*Rune Quake*, 11–12, 256

## S

sample rate, 268–269

scanners, 264

**sChainTarget**, 161–162

scoring. *See also* awards

in *Defend the Flag* with **Sigil\_Think**,  
235–236

security with **QVMs**, 151

servers, 135

server-side code, 65

sessions, players and, 95

**sfxHandle\_t**, 194

shaders, 175–184. *See also* quad shaders

for armor regeneration, 177–179,  
182–184

cgame code for initializing, 183

glowing players, creating, 179–182

quad powerup shader, 176

triple.shader, 198–199

**shotgun**

accuracy, adjusting, 58–62

damage variable, 56–58

flash of muzzle, 54

functions, 53–54

moving, shooting while, 68–72

top-down approach with, 52–53

understanding of, 53–55

**ShotgunPattern** function, 54, 63–66

**ShotgunPellet** function, 54, 56–58

sigils. *See* *Defend the Flag*

**sigilStatus\_t**, 212

**Sigil\_Touch**. *See* *Defend the Flag*

sine, 176

sin keyword, 176

size of function, 84

skinned models, 94–95

snapshots. *See* updates

*Soldier of Fortune II*, 254

sound effects

adding, 184–200

caching, 195–196

calling, 196–200

Cool Edit 2000, 269

for *Defend the Flag*, 236–238

handles for, 193–196

with household items, 268–270

mixing, 269

rewards and, 184

source

building the, 25–27

directory, 21–22

files, 22

for *Quake III*, 21

**spawn points**. *See* *Defend the Flag*

**speed**, 106

changing speed property, 107

haste rune, 106–107

sprites, 264–266

**SP\_worldspawn**, 240

static array, 157

**strcmp** function, 80–81

string compare, 80

structs, 4

for players, 96–97

pointers to, 97



switch block, 171  
 switching values, 131–132  
 system calls, 83

## T

**Targa files (TGA)**, 264  
 Adobe Photoshop, editing in, 265

### targets

of chain-lightning gun, 156  
 circle of, 88  
 for gravity wells, 83–89  
 for homing missile, 41–47

**targHeight variable**, 130

**Team Arena Expansion Pack**, 144  
 lightning effect, 157  
 ui code and, 148

**Team Arena Mission Pack, *One Flag CTF***, 207

**team-based mods**, 256–258

**team\_CTF\_blueflag item**, 218–220

***Team Fortress***, 258–259

**teammate variable**, 221

**Team\_InitGame**, 220–221, 222–223

**Team\_SetSigilStatus**, 223–224

**team variables**, 180

**TeleportPlayer function**, 90

**textured models**, 94–95

**think function**, 35, 36–37

ClientThink\_real function, 105–107  
 of *Defend the Flag* sigils, 234  
 for grenades, 76–78, 90–91

**thinking weapons**, 29

**3D engines**, 3

**3D surfaces, mip-mapping**, 195

***Threewave CTF***, 257

**time, measurement of**, 36

**top-down approach**, 52–53

**Touch\_Item**. *See Defend the Flag*

**trace\_t struct**, 164

**Transmission Control Protocol (TCP)**, 138

**trap\_EntitiesInBox**, 83–87, 248–249

**trap\_R\_RegisterShader**, 195

**trap\_Trace**, 42–43

chain-lightning gun path target, 162–165  
 with gravity wells, 83  
 ShotgunPellet function and, 56–57

**TR\_GRAVITY flag**, 75

**triangles**, 94

***Tribes***, 203

**triple.shader**, 198–199

**tripleSound handle**, 194

**triple.wav**, 198

**TR\_LINEAR flag**, 75

**trType\_t flags**, 75

**turbulence**, 184

**tvec**, 249–250

**typedef keyword**, 98

**typedefs**, 5

## U

**ui code**, 143

modification of, 147  
 responsibilities of, 147–148

**uninstalling *Quake III***, 20

***Unreal Tournament***, 203, 220

**updates**, 140–143

authoritative update, 140–141  
 pmove updates, 142

***Urban Terror***, 6–7, 202–203

locational damage system, 124

**User Datagram Protocol (UDP)**, 138

**user interface files**, 33

**user interface (ui) code**. *See* ui code

**usermd\_t**, 109, 111–112

## V

**ValidateSigilsInMap**, 250–251

**Valve Software**, 6

**variable assignment notation**, 106

**VC++**

Batch Build dialog box, 174

compiler, 5–6

projects in, 144

Service Packs for, 25

top-down approach, 52–53

using, 22–25

**VectorCopy**, 122, 170, 173

**VectorLength**, 89–90

**VectorNormalize**, 74, 122

**VectorNormalize2**, 65

**vectors**, 55–56

for chain-lightning gun, 162

tvec, 249–250

**VectorSet**, 77–78

**VectorSubtract**, 162

**velocity variable**, 70

movement and, 71

**velocity vector**, 104

**Visual C++**. *See* VC++

**Visual Studio Enterprise**, 22

## W

**Walker, Robin**, 258, 260

**WANs (wide area networks)**, 134

**Weapon\_LightningFire**, 157–160

**weapons**, 104

thinking weapons, 29

types of, 50–52

**while loop**, 44

for `Weapon_LightningFire`, 157–160

**Win32**

defined, 14

*Quake* version, 16–18

ZIP archives, support for, 261

**WinZip**, 261

**wishdir**, 122

**wishvel**, 122

*Wolfenstein 3D*, 52

## Z

**0.2f**, 73

**ZIP files**, 261

PK3 archives as, 154

## **License Agreement/Notice of Limited Warranty**

By opening the sealed disc container in this book, you agree to the following terms and conditions. If, upon reading the following license agreement and notice of limited warranty, you cannot agree to the terms and conditions set forth, return the unused book with unopened disc to the place where you purchased it for a refund.

### **License:**

The enclosed software is copyrighted by the copyright holder(s) indicated on the software disc. You are licensed to copy the software onto a single computer for use by a single user and to a backup disc. You may not reproduce, make copies, or distribute copies or rent or lease the software in whole or in part, except with written permission of the copyright holder(s). You may transfer the enclosed disc only together with this license, and only if you destroy all other copies of the software and the transferee agrees to the terms of the license. You may not decompile, reverse assemble, or reverse engineer the software.

### **Notice of Limited Warranty:**

The enclosed disc is warranted by Premier Press, Inc. to be free of physical defects in materials and workmanship for a period of sixty (60) days from end user's purchase of the book/disc combination. During the sixty-day term of the limited warranty, Premier Press will provide a replacement disc upon the return of a defective disc.

### **Limited Liability:**

THE SOLE REMEDY FOR BREACH OF THIS LIMITED WARRANTY SHALL CONSIST ENTIRELY OF REPLACEMENT OF THE DEFECTIVE DISC. IN NO EVENT SHALL PREMIER PRESS OR THE AUTHORS BE LIABLE FOR ANY OTHER DAMAGES, INCLUDING LOSS OR CORRUPTION OF DATA, CHANGES IN THE FUNCTIONAL CHARACTERISTICS OF THE HARDWARE OR OPERATING SYSTEM, DELETERIOUS INTERACTION WITH OTHER SOFTWARE, OR ANY OTHER SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES THAT MAY ARISE, EVEN IF PREMIER AND/OR THE AUTHORS HAVE PREVIOUSLY BEEN NOTIFIED THAT THE POSSIBILITY OF SUCH DAMAGES EXISTS.

### **Disclaimer of Warranties:**

PREMIER AND THE AUTHORS SPECIFICALLY DISCLAIM ANY AND ALL OTHER WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY, SUITABILITY TO A PARTICULAR TASK OR PURPOSE, OR FREEDOM FROM ERRORS. SOME STATES DO NOT ALLOW FOR EXCLUSION OF IMPLIED WARRANTIES OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THESE LIMITATIONS MIGHT NOT APPLY TO YOU.

### **Other:**

This Agreement is governed by the laws of the State of Indiana without regard to choice of law principles. The United Convention of Contracts for the International Sale of Goods is specifically disclaimed. This Agreement constitutes the entire agreement between you and Premier Press regarding use of the software.