SPELLS OF FURY

BUILDING WINDOWS 95 GAMES USING DIRECTX 2



INCLUDES 10 EXAMPLE GAMES

WAITE GROUP PRESS™

MICHAEL J. NORTON

SPELLS OF FURY

Building Windows 95 Games Using DirectX 2

Michael J. Norton



WAITE GROUP PRESS™
A Division of Sams Publishing
Corte Madera, CA

Publisher . **Mitchell Waite**
Associate Publisher . **Charles Drucker**

Acquisitions Manager . **Jill Pisoni**

Editorial Director . **John Crudo**
Project Editor . **Lisa Goldstein**
Content Editor . **Jeff Bankston**
Technical Reviewer . **Bill Heyman**
Copy Editor . **Marc Polonsky**

Production Director . **Julianne Ososke**
Production Manager . **Cecile Kaufman**
Production Editor . **K.D. Sullivan**
Cover and Interior Design . **Sestina Quarequio**
Production . **Mary Ann Abramson, Mona Brown, Michael Brumitt, Mike Henry, Brad Herriman, Paula Lowell, Donna Martin, Erika Millen, Mark Walchle**
Illustrations . **Lisa Norton, Ryan Oldfather**
Cover Illustration . **Greg Winters**

© 1996 The Waite Group, Inc.®
Published by Waite Group Press™, 200 Tamal Plaza, Corte Madera, CA 94925.

Waite Group Press is a division of Sams Publishing.

All rights reserved. No part of this manual shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, desktop publishing, recording, or otherwise without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for error or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

All terms mentioned in this book that are known to be trademarks or service marks are listed below. In addition, terms suspected of being trademarks or service marks have been appropriately capitalized. Waite Group Press cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

The Waite Group is a registered trademark of Waite Group Press, and The Waite Group logo is a trademark of The Waite Group, Inc.

Printed in the United States of America
99 98 97 96 10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data

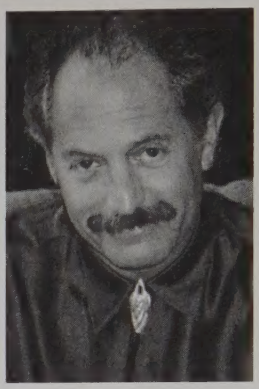
Norton, Michael.
Spells of fury : building Windows 95 games using DirectX 2 / Michael Norton.
p. cm.
Includes index.
ISBN 1-57169-067-0
1. Computer games--Programming 2. Microsoft Windows (Computer file)
I. Title.
QA76.76.C672N67 1996
794.8'15265--dc20

96-15476
CIP

DEDICATION

This is for my wife Lisa. Without her none of my projects would be possible. Thanks for the moral, artistic, and grammatical support. And thanks to my two boys, Matthew, age two, who proved video gaming is hereditary, and Ryan, age one month, who spent so many hours in my arms while I prepared this book, he learned to program.

Message from the Publisher



Welcome to Our Nervous System

Some people say that the World Wide Web is a graphical extension of the information superhighway, just a network of humans and machines sending each other long lists of the equivalent of digital junk mail.

I think it is much more than that. To me, the Web is nothing less than the nervous system of the entire planet—not just a collection of computer brains connected together, but more like a billion silicon neurons entangled and recirculating electro-chemical signals of information and data, each contributing to the birth of another CPU and another Web site.

Think of each person's hard disk connected at once to every other hard disk on earth, driven by human navigators searching like Columbus for the New World. Seen this way the Web is more of a super entity, a growing, living thing, controlled by the universal human will to expand, to be more. Yet, unlike a purposeful business plan with rigid rules, the Web expands in a nonlinear, unpredictable, creative way that echoes natural evolution.

We created our Web site not just to extend the reach of our computer book products but to be part of this synaptic neural network, to experience, like a nerve in the body, the flow of ideas and then to pass those ideas up the food chain of the mind. Your mind. Even more, we wanted to pump some of our own creative juices into this rich wine of technology.

Taste Our Digital Wine

And so we ask you to taste our wine by visiting the body of our business. Begin by understanding the metaphor we have created for our Web site—a universal learning center, situated in outer space in the form of a space station. A place where you can journey to study any topic from the convenience of your own screen. Right now we are focusing on computer topics, but the stars are the limit on the Web.

If you are interested in discussing this Web site or finding out more about the Waite Group, please send me e-mail with your comments, and I will be happy to respond. Being a programmer myself, I love to talk about technology and find out what our readers are looking for.

Sincerely,

Mitchell Waite

Mitchell Waite, C.E.O. and Publisher

200 Tamal Plaza
Corte Madera, CA 94925
415-924-2575
415-924-2576 fax

Website:
http://www.waite.com/waite

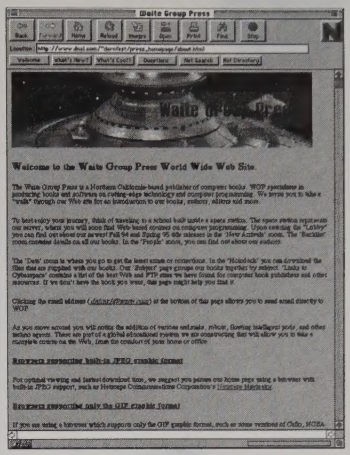
Creating the Highest Quality Computer Books in the Industry

Waite Group Press
Waite Group New Media

Come Visit

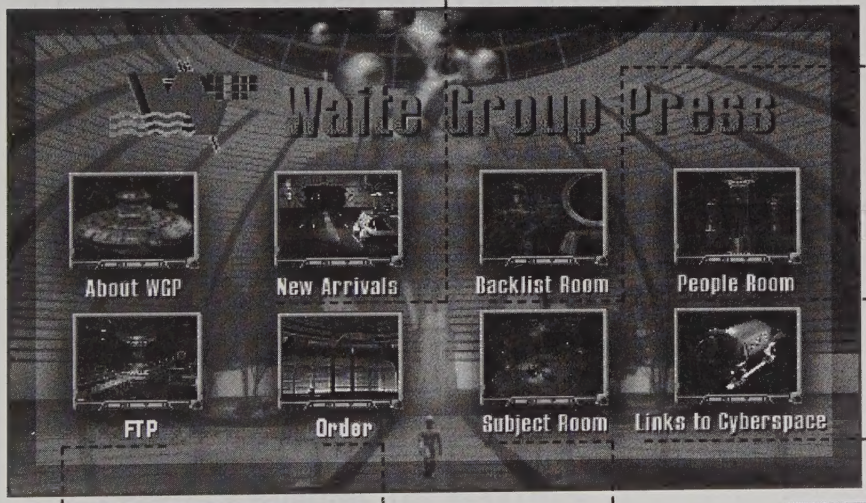
Waite.Com

Waite Group Press World Wide Web Site



Now find all the latest information on Waite Group books at our new Web site, **http://www.waite.com/waite.** You'll find an online catalog where you can examine and order any title, review upcoming books, and send e-mail to our authors and editors. Our FTP site has all you need to update your book: the latest program listings, errata sheets, most recent versions of Fractint, POV Ray, Polyray, DMorph, and all the programs featured in our books. So download, talk to us, ask questions, on **http://www.waite.com/waite.**

The New Arrivals Room has all our new books listed by month. Just click for a description, Index, Table of Contents, and links to authors.



The Backlist Room has all our books listed alphabetically.

The People Room is where you'll interact with Waite Group employees.

Links to Cyberspace get you in touch with other computer book publishers and other interesting Web sites.

The FTP site contains all program listings, errata sheets, etc.

The Order Room is where you can order any of our books online.

The Subject Room contains typical book pages which show description, Index, Table of Contents, and links to authors.

World Wide Web:

http://www.waite.com/waite
Gopher: gopher.waite.com
FTP: ftp.waite.com

COME SURF OUR TURF—THE WAITE GROUP WEB

TM



ABOUT THE AUTHOR

Michael Norton holds a B.S. in Physics, with an emphasis on Computational Physics, from San Jose State University. He began programming in 1980 on a TRS-80 model I, and was hooked from the start on video games. As a professional, he has worked the past 11 years on various machines: PDP-11s, VAXs, Apollo workstations, Macintosh, PC, and currently HP 9000 minicomputers and Sun Sparc stations. His programming experience covers the spectrum from video games, PC BIOS and hardware, IP networks, and Computer Integrated Manufacturing (CIM). Michael was a contributing author to *More Tricks of the Game Programming Gurus*, and has written for *Game Developer* magazine.



CONTENTS

PART I FOUNDATIONS OF THE ARTS

CHAPTER 1 **THE SORCERER'S APPRENTICE: WIN32 API** 3

CHAPTER 2 **DIBs: TRANSLATING THE TABLETS** 35

CHAPTER 3 **THE GATEWAY OF DIRECT ENLIGHTENMENT: DIRECTDRAW** 101

PART II THE SECRETS OF THE ARTIFACTS

CHAPTER 4 **DIRECTINPUT: INTERFACING THE ARTIFACT** 169

CHAPTER 5 **DIRECTSOUND: ENTER THE REALM OF THE QUEEN OF THE NIGHT** 221

CHAPTER 6 **TREACHEROUS PLOTS** 269

CHAPTER 7 **RECIPES FOR DESTRUCTION** 333

PART III UNLEASHING THE SPELLS OF FURY

CHAPTER 8 **ASTEROIDS** 377

CHAPTER 9 **BUBONIC THE WARTHOG** 415

CHAPTER 10 **DIRECTPLAY: NETWORKING IS THE KEY TO GLOBAL DOMINATION** 461

CHAPTER 11 **THE DUNGEON** 517

CHAPTER 12 **AGAMEMNON'S CONCISE GUIDE TO VIRTUAL REALITIES** 561

INDEX 603



TABLE OF CONTENTS

PART I FOUNDATIONS OF THE ARTS

CHAPTER 1 THE SORCERER'S APPRENTICE: WIN32 API 3

Episode 1 Agamemnon's Keeper 5

Programming with the Win32 API 7

Writing a Windows 95 Application 8

The WinMain() Function 8

WNDCLASS 11

Creating a Window 13

The Message Loop 15

The WndFunc() Function 16

Windows 95 Programming Review 19

A Generic Game Application 19

Forging a Game Console from Windows 95 19

Initializing the Game Console Window 20

The Generic Game Application's WinMain() 22

The Generic Game Application's WinFunc() 23

The Generic Game Application in Review 24

Win32 Messaging 24

Processing Game Application Messages 25

Messages 25

A Game Message Loop 27

Message Handling 28

Game Messaging in Review 30

The Dialog Box 30
A Very Simple Dialog Box 30
The DlgProc() 30
The Dialog Box Driver 32
The Dialog Box in Review 33
Conclusion 33

CHAPTER 2 DIBs: TRANSLATING THE TABLETS 35

Episode 2 Translating the Tablets 37
The Device-Independent Bitmap 39
The DIB Architecture 40
DIBs and System Memory 41
CDib: Designing a DIB Class 42
CDib Class Declaration 43
CDib Class Constructor and Destructor 43
Forging a DIB with NewDIBObject() 45
Step 1. Creating the Windows BITMAPINFO Structure 46
Step 2. Filling in the DIB's BITMAPINFOHEADER 47
Step 3. Creating a Default RGBQUAD Color Table 49
Step 4. Allocating the Image Bits Memory 50
Drawing the CDib Object Bits to the Game Screen 51
Creating a DIB in Review 52
CDib Member Functions for Reading a .BMP File 54
The Five Steps to Reading a DIB in from a .BMP File 54
Step 1. Reading the BITMAPFILEHEADER 56
Step 2. Reading the BITMAPINFOHEADER 57
Step 3. Creating the BITMAPINFO Structure 58
Step 4. Reading the RGBQUAD Color Table 58
Step 5. Reading the Image Bits 59
Reading a DIB File Format in Review 59
Reading and Displaying a DIB Using Your Generic Game Application 60
Manipulating Bits in an Offscreen Buffer 62
Understanding the Organization of DIB Bits 62
Bottoms Up or...What to Do When Your Old PCX Code Fails . . 63
Implementing an Offscreen DIB in the Windows 95 Generic Game Application 65
Manipulating Bits Offscreen in Review 67

Example 1: Creating an Offscreen DIB Object 68
Reflecting on What You've Learned 69
Sprite Rendering 69
Retrieving a Sprite from a DIB 70
Sprites of Arbitrary Shape and the Transparent Pixel 75
Retrieving a Sprite from a DIB Review 79
Example 2: Drawing Sprites to the DC 80
A CDib Object Animation Example 81
PowerSuit 81
Feasibility Study 82
Rendering the PowerSuit 90
Step 1. Calculating Sprite RECTs from Logical Grid Coordinates 90
Calculating the Patch RECT 91
Step 2. Patching Backgrounds and Rendering Sprites 95
Step 3. Sending a WM_PAINT Message 95
Step 4. Updating the Screen DC 97
Findings and Remarks 98
Conclusion 99

CHAPTER 3 THE GATEWAY OF DIRECT ENLIGHTENMENT: DIRECTDRAW 101

Episode 3 Throwing a Wrench into the Plan 103
PC Video Architecture 104
DOS Video Architecture 108
BIOS-Supported Video Mode 13h 108
Video Mode X, the Game Developer's Secret 110
Page Flipping 114
Windows Video Abstractions 116
Windows 95 Graphic Device Interface 116
Component Object Model 118
DirectDraw HAL 122
DirectDraw 123
PC Architecture in Review 123
The DirectDraw API 124
Initializing a DirectDraw Object 124
Creating a DirectDraw Object 125
Setting Cooperative Levels 127
Setting Video Modes 129

DirectDraw Surfaces 133
Creating a Primary Surface 134
Creating the Offscreen Surface 139
Restoring Surfaces 143
Blitting and Flipping Surfaces 144
DirectDraw Palettes 152
Setting a Palette Object 152
Creating a ColorKey 155
Super PowerSuit Gaming on Dynamic Terrain 158
Programming with Tiles 158
Opening a Window into the Virtual World 162
Direct Enlightenment 166

PART II THE SECRETS OF THE ARTIFACTS

CHAPTER 4 DIRECTINPUT: INTERFACING THE ARTIFACT 169

Episode 4 Interfacing with the Artifact 171
Operating Systems and User Input 173
Real Mode Input Processing 173
Interrupts 174
Interrupt Vector Table 175
DOS Interrupts and Functions 176
Keyboard Input Under DOS 177
Protected Mode Input Processing 178
Windows Messaging and User Input 179
Handling User Input Events 181
The DIDRVR Application 181
DIDRVR Animation 181
Step 1. Initializing the CDib Object 183
Step 2. Creating the DirectDraw Objects 186
Step 3. DirectDraw Instantiation 187
Step 4. Initializing the WartHog Structure 188
The Terrain Tiles 190
Bubonic the WartHog Sprites 192
Case HOG_STOP_RIGHT: 195
Case HOG_STOP_LEFT: 196
Case HOG_GO_RIGHT: 197
Composing the Display 198

Windows 95 Game SDK User Input 199
Keyboard Input 199
Keydown Message Event 199
Moving Bubonic Right on WM_KEYDOWN 200
Stopping Bubonic on WM_KEYDOWN 201
Getting Keyboard Input on Demand! 202
Mouse Input 204
Moving Bubonic with Mouse Input Messages 204
Mouse on Demand 208
DirectInput 210
Moving Bubonic with DirectInput 210
Processing Device Input with DirectInput 214
Stopping Bubonic with a Joystick Button 218
Interfacing the Artifact 219

CHAPTER 5 DIRECTSOUND: ENTER THE REALM OF THE QUEEN OF THE NIGHT 221

Episode 5 The Magic Flute 223
DirectSound Audio File Formats 227
RIFF Waveform Audio Files 227
The Interchange Format File (IFF) 228
RIFF Chunks 231
RIFF Forms 232
The WAVEFORMAT Data Structure 233
PCMWAVEFORMAT 235
Reading a RIFF File 235
The WAVE Resource 235
Unpacking the RIFF Chunks 237
Unpacking Format Chunks 242
Unpacking the Waveform Data Chunk 243
The DirectSound Interface 244
Creating a DirectSound Object 244
Creating DirectSound Buffers 246
Writing Waveforms to a DirectSound Buffer 247
Game Sound Data Structures and DirectSound 252
Using WAVEINFO Structures 253
Playing WAVEINFO Data with DirectSound 256
The WAVEINFO Driver 260

A Children's Discovery Game 261
Using DSTOOLS in an *Xplore Adventure* Game 262
Some General Scripting Required 264
Virtual Mozart 267

CHAPTER 6 TREACHEROUS PLOTS 269

Episode 6 A Dead-End 271
Plotting Pixels and DirectDraw 272
Writing to the DirectDrawSurface 273
Display Coordinates to DirectDrawSurface Translations 273
Getting a Pointer to a DirectDrawSurface 275
Writing a Pixel Using the DDTOOLS Code 276
Writing Pixels in Review 278
Drawing Lines 278
Two Points Describe a Line 279
The Slope of a Line 279
The *y*-Intercept 281
The Bresenham Scanline Drawing Algorithm 283
Distinguishing the Primary Axis and Its Slope 285
Implementing Bresenham's Line-Drawing Algorithm 288
Approximating the *y* Pixel 290
Approximating the *x* Pixel 294
Looping Through the *y* Coordinates 295
Review of the Bresenham Line-Drawing Algorithm 299
Line-Clipping Concepts 300
Line-Clipping Algorithms 301
The Cohen-Sutherland Line-Clipping Algorithm 302
Encoding the Endpoints 304
Dealing with Rejection 306
Checking for Intersections 308
Calculating Points of Intersection 310
Cohen-Sutherland Line-Clipping Algorithm in Review 313
Polygons 313
Describing a Polygon 314
Basic Transformations and Polygon Motion 316
Translation of a Polygon 316
Polygon Rotation 318
OOP—Object-Oriented Polygons 319
Building the Asteroid Ship with Our Shape Class 320

SpaceFoil: Using Polygon Objects 321
Animating with Object-Oriented SpaceFoils 321
SpaceFoil Navigation 101 326
Bring Us Around 1/3 Impulse, Mr. Sulu 328
Engage!. 330
Conclusion: Polygons and Saving the Universe 332

CHAPTER 7 RECIPES FOR DESTRUCTION 333

Episode 7 Not My Mummy 335
The Linked List 337
The List Data Structure 337
Object-Oriented Lists. 338
Designing the cList Class 338
Inserting a Node Object into a List. 341
Removing a Node Object from the List. 343
Modifying Academic Delusions into Something Useful 351
Linked Lists in Review. 354
Binary Search Trees. 354
Object-Oriented Binary Search Trees. 355
Recursion 357
BST Insertion 357
Inorder Tree Walking. 359
Binary Search Trees in Review 362
Game Program Flow. 362
Windows 95 Multitasking 363
Nonpreemptive Multitasking 363
Preemptive Multitasking 363
Scheduling Tasks and Game Events 364
Designing a Game Process Scheduler 365
State TS_SPLASH 365
TS_CRITICAL. 367
Minimizing a Game Application. 367
Maximizing the Game Application 368
TS_SUSPEND...Pausing a Game. 369
TS_TERMINATE 369
Scheduling Tasks and Game Events in Review 370
Concurrent Threads. 371
InitJoyThread() 372
Concurrent Threads in Review. 373
Conclusion 374

PART III UNLEASHING THE SPELLS OF FURY

CHAPTER 8 ASTEROIDS 377

Episode 8 *The Zero Virus* 379

Creating a Shape List 381

The Shape List Object 381

Shape Object Definitions 383

Initializing the *gameObj* Shape Object Array 384

Inserting a *gameObj* Entry into the Shape List 385

Scattering the Debris on the Screen 387

Introducing Chaos to the System 388

Processing the List Objects 390

Shape List Iteration 390

The Kinematics of Chaos 393

The Illusion of Random Disorder 393

Velocity Vectors and Quadrants 395

Collisions 396

Collision Detection Using Bounding Rectangles 396

Minimizing Collision Detection 402

Imitating Elastic Collisions 404

The Angle of Deflection 405

Concluding the Chaos 413

CHAPTER 9 BUBONIC THE WARTHOG 415

Episode 9 *The Sphinx* 417

A World Made of Tiles 418

Introduction to Tile Maps 419

The Terrain Map 420

The Tile Lookup Table 421

Rendering with Tiles 422

Non-Linear Terrains 424

Using the Slope Equation 425

beginLevel and *endLevel* Values 426

The Slope Lookup Table 427

Drawing Bubonic Using Slope 428

Scrolling Terrain Example 430

Column Slices 431

Slicing and Dicing the Logical Terrain 432

Step 1. Opening the TERRAIN.INF File 432

Step 2. Allocating the SLICE Data Structure 434

Step 3. Reading the Tiles into the SLICE Database 435
Step 4. Initializing the Base Tile Information. 436
Drawing the Terrain Slices 440
Step 1. Drawing the First Slice in a Sliding Window 441
Step 2. Drawing Slices to the DirectDrawSurface. 442
The Tile Lookup Process 443
Tiles with Animation Sequences. 444
The drawCells Driver for the TERRAIN Program. 447
Moving Objects and Items 448
Enhancing the Scenery with Moving Clouds 448
Designing a Cloud CELL Database 448
Rendering the Clouds 450
Lists and Items 452
The Items List 452
Initializing the Items List 453
Drawing the Items in the List. 456
Item Collision Detection 457
Conclusion 459

CHAPTER 10 DIRECTPLAY: NETWORKING IS THE KEY TO GLOBAL DOMINATION. 461

Episode 10 Meet Thy Maker 463
Networks and Internetworking 465
Network Layers and Protocols 467
The Application Layer 467
The Transport Layer 468
The Network Layer 468
The Link Layer 469
ARP..Excuse Me 469
User Datagram Protocol (UDP) 470
Internet Packet eXchange (IPX) 471
DirectPlay and Interprocess Communication 472
Ports. 472
Sockets 473
WinSock. 475
Blocking 475
The DirectPlay API 476
Initializing a DirectPlay Session 476
Initializing the Session Provider 477

Getting the Service Provider in Review 482
DirectPlay Client or Server? 482
Initializing a DirectPlay Server 486
Initializing a DirectPlay Client 488
A Generic Client-Server Driver for XGame 498
DirectPlay Distributed Gaming and Messaging 502
Sending a Message 502
Receiving a Message 505
DirectPlay Messaging and a Tournament Server 508
Server Passes Turn to Client 511
The Client Notifies the Server of a Completed Turn 514
Networking Is the Key to Global Domination 515

CHAPTER 11 THE DUNGEON 517

Episode 11 Escape 519
The Dungeon Map 521
A Horizontal and Vertical Scrolling Terrain 521
The Terrain Map Contents 522
Initializing the Terrain Map 524
Locating a Tile in the Map 526
Designing for Multiple Players 528
The Dungeon Player 528
The PLAYER Linked List 529
Initializing the PLAYER List 530
Initializing the PLAYER Node Data 533
Initializing Health and Armor 537
Player Movement on the Terrain Map 538
Updating the Player Position 539
Player Movement in the Distributed Environment 544
Player Characteristics 547
Player's Field of View 547
The Bresenham Line Algorithm Revisited 547
Player's Health 553
Attacking the Opponent 553
Using Potions 555
Selecting an Item to Use 557
Conclusion 559

CHAPTER 12 AGAMEMNON'S CONCISE GUIDE TO VIRTUAL REALITIES 561

Episode 12 Ground Zero 564

Mode X: The Medium of the Mystics 566

Video Mode X 566

Which Mode X? 567

DirectDraw Mode X and Video Card Memory Architectures . . 567

To Blt() to Video Memory or Not to Blt() to Video Memory . . . 569

The Mode X Renaissance 570

Initializing Mode X from DirectDraw 571

Cooperative Level and Display Mode for a Game Console Window 572

Mode X Full-Screen Cooperative Level and Display Settings . . 574

DirectDraw Flipping Surfaces for Mode X 577

Creating DirectDraw Mode X Surfaces 578

Primary Surface for a Window 578

Creating Flipping Surfaces for Mode X 581

Updating the Game Console 583

Filled Polygons 584

3D Composing on a 2D Surface 585

Convex Polygon Scan Conversion 585

Drawing the Filled Convex Polygon 590

Rendering Realities with Filled Polygons 592

Virtual I/O Stereoscopic Headset 593

Mechanics of Virtual Reality 593

Tracking the Movement of the Player's Head 594

Virtual Frame of Reference 594

Measuring YAW Displacement Using the Tracker 599

Other Considerations 600

Conclusion 601

INDEX 603



ACKNOWLEDGMENTS

My office is littered with memos and scratches on notepads of all the notes I took from the people I spoke with about the *Spells of Fury* project. If I'd had any idea this project would be so big, I would have started a database specifically for my contacts. I just hope I didn't throw away any important memos with someone's name on it who deserves credit for their work. The project took a year to complete and I spoke to a number of people in the game industry. The body of this book is devoted to giving recognition to all those who assisted me in one way or the other.

Starting from the top of the list is my wife, Lisa. A graphic designer by trade, she was drafted to be the *Spells of Fury* team chief financial officer and artist for the games and figures. Lisa is also my translator. She has an uncanny ability to take my twisted technical sentences and filter them into the English language.

The second person I need to thank from the bottom of my heart is Jennifer Thiels, for loaning me her husband, Mike, to work on this project. When Waite Group Press said they wanted ten game examples for this book, I passed out on the floor. After Lisa dumped water on my face to revive me, I picked up the Batphone and called Mike. I went to college with Mike and Jennifer. Over the years, Mike and I have worked together off and on at Activision, Chips and Technologies, and Quantum Hard Drives. Our minds have become synchronized with each other's programming mannerisms, which is a sick thought. With our sixth programming sense, I cut Mike loose on programming *PowerSuit*, *SuperPowerSuit*, *Xplore*, *Bubonic*, and *Dungeon*. Meanwhile, I took off and wrote the DirectX tools, example code, sample games, and the book. With our weekly meetings at a prominent Scottish restaurant and a feast of the number one extra value meal, we exchanged our black floppies. I wonder if the McDonald's crew thought we were industrial spies. Nevertheless, our code snapped together in the weirdest of ways.

While Mike was game designing and Lisa was cranking out the art, we brought Gregory Harsh into our *Spells of Fury* team. Gregory is the creative genius behind the *Spells of Fury* music. He is also an awesome 3D artist whose skills I'm going to exploit in my next project. Gregory wrote the *Asteroids*, *PowerSuit*, *Dungeon*, and *3DTomb* scores. My only instructions to Gregory were, I needed something spacey, something *BattleTech*-like and something *DOOM*-ish. By the end of the work week, I received several diskettes labeled, *DOOM*-ish, *Robot*-like, and *spacey*. Gregory Harsh, you're awesome!

The Waite Group Press side of the *Spells of Fury* team included Aquisitions Editor Jill Pisoni, and Project Editor Lisa Goldstein. Completion dates got pretty rocky there for a while. The SDK was going through development changes and the technology wasn't readily available to the public. I think Jill and I both drove up the stock price of Tums by our mass quantities of consumption.

Lisa Goldstein, the Project Editor, was the light at the end of the tunnel for *Spells of Fury*. To this day I am still amazed at the motivational skills Lisa exercised on this project. There were days where I climbed out on the ledge of my single-story house ready to jump and she talked me down. Thanks for changing my perspective on the project during my personality changes.

Alex St. John, at Microsoft, was a critical player in helping make my contacts. You went out of your way for me and I really appreciate that. A special thanks to Miriam Mueller as well, and all the others up at Microsoft.

Marcus Andrews provided for me the holy doctrine of DirectInput. Marcus, thanks for all your advice and help with DirectInput. Roger Weiss, the creative genius behind the *DOOM95* DirectPlay interface, entertained my stupid questions. Roger, you were instrumental in getting me on the right track. Thanks.

Fred Humel, the programmer of *DOOM95*, and Rick Johnson at Raven Software, I can never thank the two of you enough for giving me help in a timely manner. Rick, your ability to pinpoint problems in code is uncanny! As for Fred, you're awesome. I'm going to shave my head and beat a tambourine at the San Jose International airport in your reverence.

Ken Nicholson, Director of Game Technologies at ATi, thanks for supplying me with the Mach64 and 3DRage video cards I needed for this project. Ken, we're going to have to make a *Bubonic vs. Bubsy Deluxe Edition* to your *Super Bubsy*.

Joe Tretnic at S3 was instrumental in providing me with the S3 development kit and video card. One thing that wasn't immediately apparent with DirectDraw was that the developer had to be familiar with all card architectures. Thanks Joe, your dev. kit saved my hide when it came to working with bank-switched architecture.

Matt Coil and John Alper of Exos, what can I say? We put together the *Spells of Fury* Chapter 12 that never was. Matt Coil, the creative genius

behind the Exos force feedback joystick API, gave *Bubonic the WartHog* some new interesting features. My favorite was the stick's jerking motion as Bubonic landed in the spikes. Hopefully the shroud of secrecy will be removed in time for the next project. Thanks John and Matt.

Todd Rowen and Ken Hawkins of Virtual I/O, thanks for the timely response. Due to the constraints of DirectInput 1.0, we couldn't get the i-glasses to function with DirectInput. However, DirectInput 2.0 is in the works for bidirectional communication with stereoscopic headsets. Your Win32 API worked just fine and allowed me to leave DirectInput for the game controller. 3DTomb looks great through the i-glasses.

Barbara of Public Relations at PowerSoft is forever on my Christmas list. Thanks for being understanding and eager to help me with *Spells of Fury*. Watcom is an awesome compiler. But hey, you don't have to convince anybody in the game industry of that.

Forte Technologies, this was a relationship established too late for *Spells of Fury*. The VFX1 headset is awesome and I know exactly how I'm going to put it to use in my upcoming project.

A special thanks to the *Spells of Fury* test team, Matthew (who spent more time testing Fred Humel's code than mine), Ryan Norton (the one-month-old kid who forced me to type with one hand), Meggan and Jad Thiels (the kids forced to code in Mike's game sweatshop), and Bobo the Wonder Dog (who painstakingly tracked down my page faults).

Countless others assisted me in more ways than I can possibly thank. If I forgot to mention your name, I apologize. But the one person I really need to thank is Mitch Waite for having the staff to make a project like this work. No other publisher can do what your people do, and that's why Waite Group Press puts out the best books.

I know exactly what I am going to do for a follow-up to *Spells of Fury*, but the wife says I gotta take three months off. After that I put the insulated electrical gloves on, the white lab coat, and pull down the welding goggles; Skunkworks will be operational. See ya in 365!



FOREWORD

It was almost two years ago that I became Microsoft's game technology evangelist. I'd like to say that I was the first, but there was one before me, a poor fellow named Paul Bloom who had nothing more than Video for Windows and the MCI extensions to hold up to developers as a compelling case to make Windows games. Needless to say, his fortune as a game evangelist was not as good as mine has been. I was very lucky to have two very sharp friends working with me in Microsoft's Developer Relations Group who were eager to do something extraordinary, and I was fortunate to have become the game evangelist at a time when Microsoft was making a tremendous transition to an all new operating system technology. The conjunction of these two events made it possible for Windows to take a radical new direction in interactivity.

We named our little effort the Manhattan Project, in part because we thought it was cool, but primarily because it really annoyed our management who were terrified that if the name leaked out, hordes of nuclear protesters would picket the company. The gaming press, of course, loved this and used the nuclear theme heavily in their first write-ups on the technology.

The delivery of the technology had a profound effect on Microsoft and on the gaming industry. Suddenly Windows wasn't just for word processors and spreadsheets anymore. A vast new array of applications and possibilities became available to developers. Windows 95 had made enormous strides towards consumer friendliness, including things like automatic hardware configuration detection and AutoPlay for automatically launching applications from a CD. Coupled with the new multimedia power enabled by the Game SDK, it was finally possible to write highly interactive PC applications that human beings who didn't know their IRQ settings or how to edit their config.sys could actually install and use. Further, the SDK had accomplished something almost magical in software terms. It made it possible to write multimedia applications that could actually run faster, with a thin layer of

software indirection under Windows, than going directly to the hardware under MS-DOS.

It shortly became clear to us that although we had set out to create a set of APIs specifically for accelerating games, we had really created a new approach to API and driver architecture that provided the benefits of device independence, while preserving low latencies and real-timeliness. Although, specifically, very useful for games, the SDK also represents an API design philosophy that we decided was distinct enough to warrant its own naming convention.

All of the Game SDK APIs are called Direct something. We have DirectPlay for networking, DirectInput for joysticks, DirectSound for audio, and DirectDraw for graphics. The family of APIs is called DirectX. DirectX APIs are characterized by the property of being very thin and close to the hardware. They come as close as possible to directly reflecting the functionality of the hardware while preserving device independence. Direct APIs never hide what functionality is hardware-accelerated and what functionality is not. Although we provide software emulation for any hardware feature DirectX supports, we do not hide the emulation from the application. The application can always choose to do its own thing if it finds that something it needs from the hardware like a transparency blitter isn't implemented on a particular card. This puts tremendous power and flexibility in the hands of the application, while keeping device independence. These are the specific properties that make an API "Direct."

We're also very pleased to see the first books on DirectX getting published for a much larger developer audience. Michael Norton came to us very early in the Game SDK development effort to talk about doing a book on the technology. His book is definitely the most comprehensive work on the technology that I've seen to date. I recall that we insisted it would have to have a mushroom cloud on the cover in order for us to cooperate, but our hearts softened on that point and we ended up working with Michael periodically despite his publisher's unwillingness to support a nuclear theme. Michael even taught a six-hour course for us on using DirectX at the most recent Computer Game Developers Conference, for which I was so grateful that I agreed to spend my Sunday afternoon writing a foreword for his book.

Needless to say, everyone on the Manhattan Project hopes you use the technology to make awesome interactive applications, and get as rich as Bill selling them.

—ALEX ST. JOHN



INTRODUCTION

In the spring of 1995, I completed a writing project and decided to tackle another project that would be of interest to game designer hobbyists as well as professional developers. My first thought was of publishing a technical document on how to program a 3D game engine such as that used in *DOOM*. That year I attended the Computer Game Developers Conference to schmooze with developers and gain an understanding of what would be of interest to them at their level. One of the topics offered at the conference, for the early birds, was a presentation on the new Microsoft Game SDK. While taking my seat I thought, "This is going to be a *cute* little presentation on the *son of WinG*." I heard developers near me sharing the same preconceived notion.

Our attitudes changed when Alex St. John, the Game Evangelist for the Microsoft Game SDK, demonstrated the *Immortal Klowns* application using the new technology. Granted, *Immortal Klowns* is not a significant application in the halls of gaming fame. It did, however, demonstrate that the Microsoft Game SDK was capable of thrusting the Windows 95 operating system into the arena of serious game development platforms for the PC. At that time, DOS was the premier operating system for commercial games. Frame-rates on the Microsoft Game SDK demos exceeded the capabilities of games designed for DOS. I was immediately skeptical. It had to be some software smoke and mirrors trick, ...but it wasn't. I then realized that DOS was one *DOOMed space marine*.

In that moment of revelation, I determined that my next writing project would focus on this new next generation of gaming technology. You now hold the compilation of my research on the Microsoft DirectX Game SDK, *Spells of Fury*. The format of this book is somewhat different from other programming books. Programmers thrive on science fiction and fantasy to feed the creativity of binary realizations. Thanks to the flexibility of the Waite Group Press, I was allowed to proceed down an avenue mixing creative science fiction with technical game programming fact.



ORGANIZATION OF THIS BOOK

The book is foremost a collection of professional techniques I pulled in from industry programmers. The style in some of the code tends to change because I wanted to preserve most of the concepts that were presented to me. The content of *Spells of Fury* is divided into three sections, Foundations of the Arts, The Secrets of the Artifact, and Unleashing the Spells of Fury. Each chapter is designed to be a building block to the next chapter. The first five chapters will put together most of the modular tools using the Microsoft DirectX Game SDK to rapidly prototype virtually any style of game.

PART I. FOUNDATIONS OF THE ARTS

I rewrote Chapter 1, *The Sorcerer's Apprentice: Win32 API*, three times as I pulled in information from developers. My original chapter contained MFC code, but Alex St. John and Robert Hess slapped my hands with a wooden ruler and sent me through their Windows reform school. The final form of Chapter 1 introduces Win32 programming and begins a dissertation on how to shut off Windows attributes that interfere with game applications. These first three chapters are the building blocks for game applications presented throughout the book. Chapter 1 introduces a generic DirectX Win32 application that will be used over and over to prototype new games.

Chapter 2, *DIBs: Translating the Tablets*, contains a complete discussion on DIBs and the GDI. I presented a self-contained CDib class, using no MFC, to provide access to image bits and palettes for transferring images to DirectDraw. The CDib object is used in a GDI-based game example, PowerSuit, at the end of the chapter.

Chapter 3, *The Gateway of Direct Enlightenment: DirectDraw* is, you guessed it, DirectDraw. This chapter investigates how a DirectDraw object encapsulates the functionality of the video device. You'll learn how to manipulate video memory, called a DirectDrawSurface, and harness the power of DirectDraw. At the end of Chapter 3 you create a 3/4 view *Super PowerSuit* game using DirectX technology and page flip from within a Windows 95 application.

PART II. THE SECRETS OF THE ARTIFACT

This section reveals industry secrets on how to turn an operating system, designed for word processing and databases, into a powerful gaming platform.

Chapter 4, *DirectInput: Interfacing the Artifact*, examines how to get around the annoying problems associated with Windows message-based input. Here I discuss keyboard and mouse input for the most demanding of game applications. This chapter also introduces the new DirectInput API that communicates directly with the game controller(s).

Chapter 5, *DirectSound: Enter the Realm of the Queen of the Night*, reveals how to work with WAV files, how to parse them, and of course, how to create a DirectSoundBuffer to play these sounds. This discussion includes streaming buffers. Chapter 5 concludes with a discovery game constructed from all the DirectX components and tools designed so far.

Chapter 6, *Treacherous Plots*, reveals the methods of scanline conversion and how to manage a DirectDrawSurface. Vector animation, such as that used in *Asteroids*, *BattleZone*, *SpaceWar*, and *Tempest* is the subject of this chapter. At the end of the chapter these techniques are used to construct the *SpaceFoil* space war game.

Chapter 7, *Recipes for Destruction*, provides a quick reference for writing linked lists and binary trees used in the games that follow this chapter. Chapter 7 also demonstrates the advantages of the Windows 95 32-bit multitasking operating system. It also discusses preemptive and non-preemptive multitasking, threads, and minimizing application examples.

PART III. UNLEASHING THE SPELLS OF FURY

The last five chapters demonstrate different games that can be assembled using the tools from the previous chapters of *Spells of Fury*. Section III is dedicated to revealing the secrets of different style video games from *Asteroids* to *DOOM*.

Chapter 8, *Asteroids*, using the techniques explained in Chapter 6, recreates the arcade classic using DirectX. Here simple physics creates a chaotic debris field.

Chapter 9, *Bubonic the WartHog*, shows how you can make your own console style game. Examples demonstrate how to create tile lookup tables to animate onscreen and offscreen objects.

Chapter 10, *DirectPlay: Networking Is the Key to Global Domination*, explains how to create games on local area and internetworks. Configuration of DirectPlay players and a server are covered. The example game, *NetChess*, shows a game client-server architecture.

Chapter 11, *The Dungeon*, takes interactive M.U.D.s to the next level. This game example shows how to construct a real-time DirectPlay game application. Learn how to update multiplayer games with animated sprites over a network.

Chapter 12, presents *Agamemnon's Concise Guide to Virtual Reality*. This final project involved some of the most brilliant programmers in the industry today. Find out how the top game programmers are using DirectDraw. This example includes a simple *DOOM*-style engine and a Virtual I/O i-glasses stereoscopic headset interface.

This book is all about creativity, programming, and plain old fun. I hope you have as much fun modifying and experimenting with the applications as the *Spells of Fury* team did in putting them together. This project was a blast!

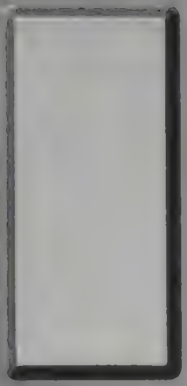


The *Spells of Fury* CD-ROM

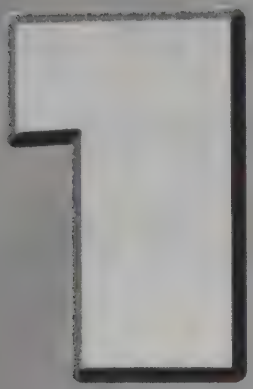
The code for the chapter examples in *Spells of Fury* is located in the master directory \FURY, located in the root directory of the CD-ROM. Subdirectories are categorized by chapter name. Chapter 1 examples are found in the subdirectory \FURY\CHAP_01. If you so desire, you may drag the entire \FURY directory over to your hard drive or copy one chapter at a time. Check the resources available on your hard drive before doing so. These examples contain art and .WAV sound files.

The build dependencies of the Visual C++ examples require a \FURY directory with a \DXTOOLS directory and the \CHAP_0X\EXAMP0X paths intact. All the examples provided were compiled using both Watcom C++ 10.6 and Visual C++ 4.1. Executable examples are provided in each chapter directory under the subdirectory \CRYPT. A NOTES.TXT file is provided in the \CRYPT directory defining the application build dependencies in all chapters except Chapters 1, 2, 10, and 11. This file is useful if you're a Watcom coder and rely on your own makefiles and coding tools rather than the Microsoft Developer's Studio.

If it's late at night and your eyes are tired but your body is wired on caffeine, play with *DOOM95*. Its path from the CD-ROM's root directory is \STUFF\DOOM95. A zipped version is located in \ARCHIVE from the root directory. Now, go forth and create great DirectX games!



FOUNDATIONS OF THE ARTS



THE SORCERER'S APPRENTICE: WIN32 API



THE SORCERER'S APPRENTICE: WIN32 API

This book is for the DOS and Windows 95 developer who wants to quickly migrate over to the Windows 95 Game SDK. The book is designed to be a hands-on guide to developing game applications using the Windows 95 operating system. The code found in the following chapters is modular. In this chapter, you'll learn how to write a Windows 95 32-bit application. You'll also learn how to design a generic Windows 95 game application that you will use over and over to snap in various Windows 95 Game SDK libraries. By Chapter 5, you will have developed enough modules to rapidly prototype just about any style of game using the Windows 95 Game SDK.

Before you can unleash the computing power of Windows 95 for game development, you will first have to learn how a Windows 95 application functions. In Chapter 1, I present to you a complete overview of writing a Windows 95 game application driver.

EPISODE AGAMEMNON'S KEEPER



The last thing I remembered doing that morning was surfing at a state beach in Santa Cruz, California. I remembered being yanked off my board and struggling under water. I had thought I was being attacked by one of the great white sharks that are indigenous to the Northern Pacific coast. That was how my morning began. When I came to, I didn't know what the current time was, or even whether it was day or night. At that moment, I was just thankful that I hadn't been breakfast for a 25-foot long fish.

My head had a nasty tingle, probably from a drug used by my abductors when they nabbed me. I could feel a cold slab of concrete my body was lying on. My vision started to improve a little. I could see soldiers leaving the room I was in. A solid, polished steel door closed behind them.

I turned my painfully throbbing head to look around. In the middle of the room was a large sarcophagus, like an Egyptian artifact. It was huge. Thousands of wires and relays fed into the artifact. I could also make out a figure walking towards me. As the figure came closer my eyes focused, and I could see it was a woman in her late twenties, early thirties, red hair, green eyes, and professionally dressed, wearing a white lab coat.

"Are you all right? What have they done to you?" she asked.

"Leave him be," the other shadowy figure ordered. "Mr. Maelstrom and I have some unfinished business to settle before we make any formal introductions." In the shadows, I could see an orange glow, like someone was taking a drag on a cigarette. The smell of cigarettes repulsed me.

The shadowy figure, who was smoking, moved toward me. As he walked toward me into the light, I could see he was dressed in a dark suit, white shirt, and a black tie that was loosened at the collar. The man appeared to be in his late fifties, but the years of smoking made him look even more haggard. The man kneeled down to me to help me to my feet. His yellow, soulless eyes squinted at me, trying to intimidate me. He took a long drag on his cigarette and blew a ring of smoke in my face. I gagged.

"So, Bary Maelstrom, the super-hacker who has broken all my codes and stolen all my secrets. Do you realize how much trouble you have caused for me? I didn't sleep for a week when you retrieved the Majority 12 (MJ-12) files. It took a lot of doing to discredit you. Little did the general public know that you really did have the actual files on the little green men from Mars," the smoking man continued.

"Well, according to your files, they aren't green!" I disputed.

The man threw down his cigarette and stepped on it to extinguish it. It was an obvious gesture of body language that he would do the same thing to me if I didn't cooperate.

"Well, obviously I have something you want. Or else you would have extinguished me too while you had the opportunity," I refuted.

The man walked into the shadows once more, but this time to find a light switch. Light flooded the room and my pupils with a painful gush of sensory overload. I shook my head and rubbed my eyes. The entire room now came into focus.

"Bary Maelstrom, meet Agamemnon. This is artifact x-001. I presume you are familiar with its history by now." The dark-suited man walked over to the Egyptian artifact.

"But I thought this was just UFO folklore, like the Philadelphia experiment," I replied in awe. "Is it an alien artifact?"



"No, it's definitely from this world. As you can see from the markings, it's obviously Egyptian. My prized scholar here, Dr. Kala McTaggert, believes it predates the exodus of the Israelites out of Egypt," the grim looking man continued.

"So why is it here? Wouldn't the Smithsonian love to get their hands on this?" I asked.

Dr. McTaggert entered the conversation. "This artifact is a computing device with phenomenal capabilities. It's clearly superior to anything we've got. When it was seized from the Nazis in World War II, this device was running computations to enhance V-2 propulsion systems to reach North America. We also have reason to believe it was computing a means to enrich uranium on a simpler scale than is in use even today," Kala informed.

"That's quite enough, Kala." Kala's superior interrupted her briefing.

"So, that doesn't answer my question. Why isn't this in the Smithsonian?" I asked once more.

"Two reasons Mr. Maelstrom, which leads into why you are here. One, the device is sentient. The creator of this device, whom ancient scripts identify as a sorcerer named Agamemnon, somehow downloaded an image of his mind into the artifact. Second, your predecessor, Dr. Goldmen, put a little virus in Agamemnon. A virus you helped her design." The man paused for a moment.

"You must have given her a good reason to activate the zero contingency virus. That was meant to effectively stop any devices participating in an aggressive act on American soil. Namely a terrorist group trying to produce or activate a fissionable device. If Dr. Goldmen unleashed that on Agamemnon, then so be it. I won't help you."

"Bary." The man paused to collect his words. "The artifact is tied into the defense net. It has been since the 1950s. It was the first computational device used by the government to counteract an all-out nuclear attack during the cold war. There's no telling what the artifact will do if it thinks it's being attacked by its own side. That thing is interfaced to all of our birds and knows the codes. This was the only computer the Soviets couldn't interface. You stop that virus and you stop it now!" the man ordered.



Programming with the Win32 API

In this chapter, you will cover the fundamentals of a Windows 95 application. If you're a DOS programmer wanting to learn the Win32 API, you will examine how to construct a Windows 95 application step by step. If you're a Windows 3.1 programmer wanting to cross over to coding 32-bit Windows 95 applications, a lot of these concepts will look familiar to you.

For a DOS programmer migrating over to the Windows 95 platform, there are subtle differences in the operating system that the developer needs to be

aware of. The first and foremost difference is that there is no direct level access to ports. This doesn't mean you can't communicate with the ports; it just means that your current DOS methods of hardware communications have changed completely. Win32 is a very powerful API with hundreds of routines and, taken together with the Microsoft Game SDK libraries, you will find that you can abandon your old DOS libraries. This was an emotional turning point for me when I started coding Windows programs a few years back. There's a time to let go of things and I realized it was time to let go of the mountains of code I had written for DOS. All the utilities I wrote for DOS are available for Windows 95, whether they're in the Win32 API or the Microsoft Game SDK.

The first thing you have to do to become a Windows 95 programmer is to break your old coding habits. If you're like me, a decade of old coding habits is hard to break. A DOS programmer typically writes an application to tell the MS-DOS operating system to perform a certain task. This task could be to configure registers of a video card at a selected port in the VGA. A Windows 95 application works differently. It is the Windows 95 operating system that tells your application when to perform a task. A Windows 95 application waits until it is sent a message by the operating system. The message is received by a special function in your application and this function will take action on the message. You will use the Windows 95 and Win32 APIs to respond with different actions to a message.



Writing a Windows 95 Application

Another old habit you need to break is how you code a Windows 95 application. Typically when you start to play around with a new language, you always code the classic *hello world* application. For some reason, this seems to be a major confidence builder for every programmer. It's kind of like a coding ice breaker. The new API always seems so foreign until you write this simple little application. And suddenly, once you get this little program running, you find yourself diving deeper into the APIs that support the language.



The WinMain() Function

In C and C++, you use a special function classically referred to as a program entry point by the C language designers Kernigan and Ritchie. But you know this function as main(). This is the minimal amount of code you need to write any C or C++ application. The code presented in this book is all in C++. Windows 95 takes advantage of a lot of object-oriented concepts. To use this book, you need to have a programming background in C or C++. In C++, your program entry point for a DOS program looks like this:




```
#include <stdio.h>
int main(int argc, char* argv[])
{
  printf("hello world\n");
}
```

This is your simple *hello world* application, which you first wrote eons ago. I think I first keyed this in on a teletype for a DEC PDP-8 computer in my high school computer class. You're going to resurrect your old code and put it to use to demonstrate the Windows 95 entry point function, WinMain(). Just as every C and C++ application requires a main() function, every Windows application requires a WinMain(). You can use WinMain() to write a simple application such as *hello world*, as shown in Listing 1-1:

LISTING 1-1 Example 1: A Simple Windows 95 hello world application

```
#include <windows.h>

int WINAPI WinMain (HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow)
{
  MessageBox(hwnd,"Hello World", "Spells of Fury", MB_OK);
  return 1;
}
```

The Windows 95 application looks similar to the DOS counterpart with considerable changes. Note that you include the header file WINDOWS.H rather than STDIO.H as you would in a DOS console-style application. This doesn't mean you'll never use the header STDIO.H. The point here is that Windows I/O is provided primarily through the routines declared in the WINDOWS.H header file.

The application demonstrated here only generates a Windows message box that displays the *hello world* string. When the message box terminates, so does your application. This application doesn't create a message loop for message handling, nor does it create a window. This is some of the added functionality you need to add to your Windows 95 application.

Another visible difference between the C++ style main() and the Windows API WinMain() is the arguments passed to each function. As has been demonstrated, both style applications call their respective main() functions from the application start-up code. When a Windows 95 application begins execution, four parameters are passed to WinMain(). The first argument passed to WinMain() is the instance handle, type HINSTANCE, *hThisInst*. The parameter *hThisInst*, synonymous to a task ID, uniquely identifies this particular instance of the program executing. Windows allows multiple executions of a single program, called an instance. For example, multiple DOS windows can execute on Windows 95. Each DOS window is assigned a unique instance value. The *hThisInst* is the current instance of the program.

The second parameter passed to WinMain() is also an instance handle, *hPrevInst*. This is the instance handle of the most previous instance of the same application executing. Older Windows applications were designed to be process-based multitasking programs. Windows 95 uses threads and processes. In Windows 3.1, when a second DOS window was opened, the new DOS window assigned its *hPrevInst* value a non-zero value. Under Windows 95, a single program may multitask with two separate instances executing simultaneously; hence the *hPrevInst* will be zero.

The third parameter used by WinMain() is *lpszArgs*, which is a pointer to the command line arguments. The *lpsz* is a common naming convention in Win32; it's a 32-bit pointer to a null-terminated string.

The fourth and last parameter is *nWinMode*, which is simply a display attribute of the window when it executes. This value can be SW_SHOWNORMAL or SW_SHOWMINNOACTIVE, as defined in the WINDOWS.H header file. This argument is used to launch the program as a normal displayed window or an initially minimized window.

As your adrenaline pushes you to examine more Windows code, you'll discover there are actually several different styles of WinMain() declarations, just as there are different C++ styles of main(). Figure 1-1 shows the different coding conventions that span the evolution of the Windows and Win32 APIs. Older Windows 3.1 applications use the int PASCAL WinMain or int FAR PASCAL WinMain() prototype. The Win32 API recommends using the int WINAPI WinMain() prototype. Both styles will compile without any problems.

Before you move on and add more functionality to your Windows 95 application, I do need to make DOS programmers aware of the Windows variable declaration and coding conventions used by Windows programmers. The WINDOWS.H header file has TYPEDEFed some common variables, such as unsigned int to UINT, and unsigned long to DWORD, and char to BYTE. I found the nomenclature a little confusing when I first started coding Windows applications.

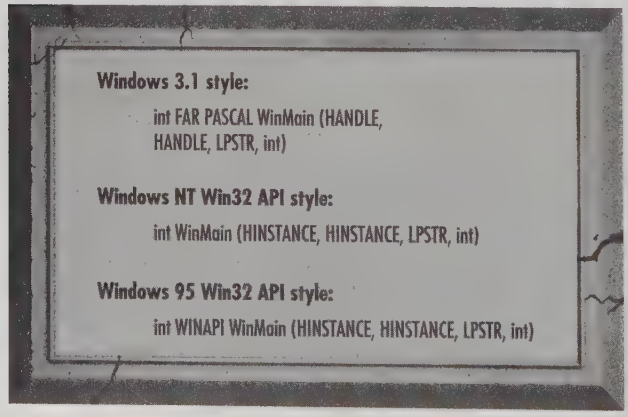


FIGURE 1-1
Windows API WinMain() function coding conventions

Your hello world application was great for a confidence builder, but it really doesn't do anything. You didn't display a window and you didn't start a message queue to interact with the Windows 95 operating system. Now take the next step and give your Windows 95 application a window to display.



WNDCLASS

To create a window for your Windows 95 application to display, you need to initialize a window class. The window class, a TYPEDEFed structure WNDCLASS, in the Windows API, contains all the attributes and behaviors of the window you want to display. The WNDCLASS, shown in Figure 1-2, is typically the first structure initialized by the programmer in the WinMain() function. The most important field of WNDCLASS is the *lpfnWndProc* field. The *lpfnWndProc* field is the address of your message handling function that will process the operating system messages for this display window. The WndFunc() function used by your application for processing messages will be introduced shortly.

The Windows APIs require that the WNDCLASS fields *lpszClassName*, *lpfnWndProc*, and the instance handle *hinstance*, be initialized by the programmer. The other behavior values, such as the cursor or menu, can use default attributes. When all the desired fields have been defined, the WNDCLASS structure then needs to be registered with the RegisterClass() function call. The code to initialize your WNDCLASS is shown in the following example:

```
static char szAppName[] = "Example 02 Spells of Fury";
WNDCLASS  wndclass;

wndclass.style          = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc    = NULL;      // window function
wndclass.cbClsExtra     = 0;         // no extra count of bytes
wndclass.cbWndExtra     = 0;         // no extra count of bytes
wndclass.hInstance      = hThisInstance; // this instance
wndclass.hIcon          = LoadIcon (NULL, IDI_APPLICATION);
wndclass.hCursor        = LoadCursor (NULL, IDC_ARROW);
wndclass.hbrBackground  = GetStockObject (WHITE_BRUSH);
wndclass.lpszMenuName   = NULL;
wndclass.lpszClassName  = szAppName;

RegisterClass (&wndclass);
```

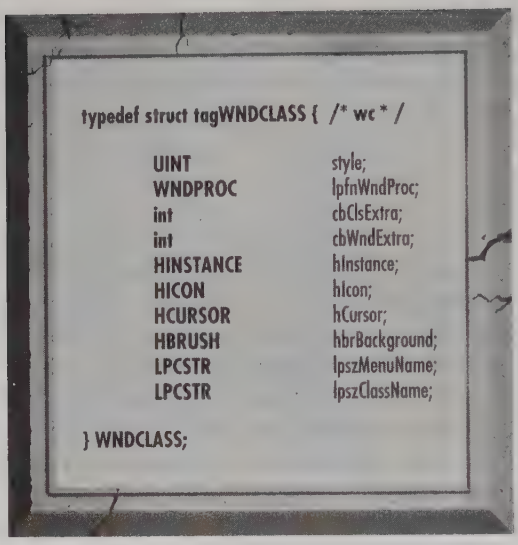


FIGURE 1-2
TYPEDEF of the WNDCLASS structure

This is pretty close to the code you will use to initialize the display window for your game applications. The exception here is that you have left the address to *lpfnWndProc* NULL. You have not instructed your WNDCLASS as to what function will process your application's messages. This is not a very good Windows programming practice. We will correct this when we discuss the WndFunc() function.

The WNDCLASS style field defines the window's style. For your application, you are using the classical Petzold window style with the value CS_HREDRAW|CS_VREDRAW (read as CS_HREDRAW OR CS_VREDRAW). CS is an identifier in WINDOWS.H for class style. The bitwise OR states that this particular window class is to be redrawn whenever the horizontal or vertical size changes. In actuality, this field can be zero, since a Windows 95 Game SDK application will be executing full screen in 640 x 480 resolution. However, there may be a case where a user would want to resize the window.

The WNDCLASS *lpszMenuName* field is left NULL. When your WNDCLASS is registered, you will have no Windows style pull-down menu associated to your window. Your intent is to write video game applications, not spreadsheets or word processors. You want to provide a different feel to your menus to help pull the player into your creative applications. The game menus need to have the same feel as a Sony PlayStation or Sega Saturn application. The blue and gray pull-down system menus really don't provide this. So, you'll turn them off.

The WNDCLASS fields *cbClsExtra* and *cbWndExtra* are initialized to zero. These fields are used if the WNDCLASS needs to allocate extra memory. The variable prefix *cb* is a common Windows naming convention for count bits.

The background color of the window is specified by the field *hbrBackGround*. Like all other lowercase prefixes in Windows, *hbr* has a special meaning, handle to a brush. The brush is the pattern of pixels used to draw the window

background. In this case, a function call to GetObject(WHITE_BRUSH) returns the classic white Windows background. The white background just provides some satisfaction when executing the sample generic game application window. When all of the fields of WNDCLASS have been initialized, the application registers the window class with a call to RegisterClass().



Creating a Window

When the WNDCLASS has been initialized and registered, the application can then create a window. The Win32 API routine CreateWindow(), shown in Figure 1-3, is used to initialize a window based on the attributes you defined and registered in the WNDCLASS.

The Win32 API call CreateWindow() can create several styles of windows for us. You can create an overlapped window, which is the standard window with a caption and a border. You can create a popup window, a child window, a minimized or maximized window, and so on. This is where you specify how you want your window initially displayed. To create console style games using the Windows 95 Game SDK, you will want to use an overlapped window. To initialize the window for your application you use the following code:

```
HWND   hwnd;
hwnd = CreateWindow (wndclass.lpszClassName,// window class name
       "Spells of Fury: Creating a Window",   // window caption
       WS_OVERLAPPEDWINDOW,              // window style
       CW_USEDEFAULT,                    // initial x position
       CW_USEDEFAULT,                    // initial y position
       640,                              // initial width size
       480,                              // initial height size
       NULL,                             // parent window handle
       NULL,                                    // window menu handle
       hThisInstance,                    // program instance handle
       NULL);                            // creation parameters

ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);
```

The CreateWindow() function requires the name string of the registered WNDCLASS that is associated to the window you want to create. The other values in the function call may use the Windows default macro CW_USEDEFAULT. You use this macro to tell Windows where you want the upper-left of your window to be placed. For the window height and width you explicitly requested the dimensions of 640 x 480 pixels. This will be the size of the game display. Your window does not have a menu so you also pass in NULL for the menu handle. You do need to specify the handle of the application that is creating the window. Here you will pass in the instance handle HINSTANCE. For the last field, you also pass a NULL value. If you were to create a client window for your application, you would pass in the address of where the data describing the client window could be found.

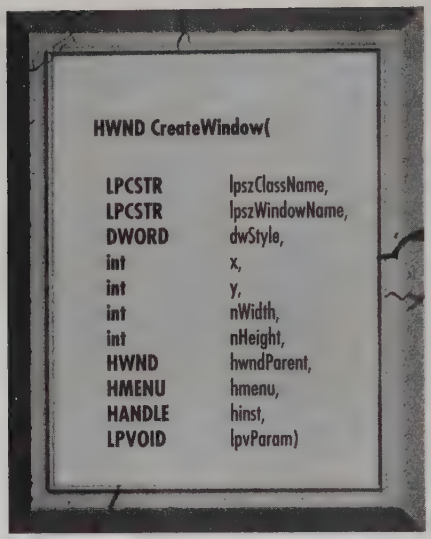


FIGURE 1-3
Win32 API call to create a window

When the Win32 call CreateWindow() has completed, it will return a handle (HWND) to the newly created window. When a window is created it is not displayed; you must call the Windows API function ShowWindow(). The call to ShowWindow() requires two arguments to display the window. You must give the ShowWindow() function the handle HWND that was returned from the CreateWindow() and you must provide a Windows macro to describe how you want the window to be shown. These ShowWindow() (SW_) macros include SW_MAXIMIZED, to display the window maximized, and SW_MINIMIZED, to display it in a minimized state.

If the ShowWindow() call is successful, the WS_OVERLAPPED window should be displayed on the screen. The code you used in Example 2 to create a window is shown in Listing 1-2.

LISTING 1-2 Example 2: Creating a window for Windows 95

```
int WINAPI WinMain (HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow)
{
        static char szAppName[] = "Example 02 Spells of Fury";
        HWND hwnd;
        WNDCLASS  wndclass;

        wndclass.style = CS_HREDRAW | CS_VREDRAW;
        wndclass.lpfnWndProc = NULL;              // window function
        wndclass.cbClsExtra = 0;                  // no extra bytes
        wndclass.cbWndExtra = 0;                  // no extra bytes
        wndclass.hInstance = hThisInstance;       // this instance
        wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
        wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
```

```
    wndclass.hbrBackground = GetStockObject (WHITE_BRUSH);
  wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = szAppName;

    RegisterClass (&wndclass);
    hwnd = CreateWindow (wndclass.lpszClassName,
           "Spells of Fury: Creating a Window",   // window caption
           WS_OVERLAPPEDWINDOW,                   // window style
           CW_USEDEFAULT,                         // initial x position
           CW_USEDEFAULT,                         // initial y position
           640,                                   // initial width size
           480,                                   // initial height size
           NULL,                                  // parent window handle
           NULL,                                  // window menu handle
           hThisInstance,                         // instance handle
           NULL);                                 // creation parameters

    ShowWindow (hwnd, nCmdShow);
    UpdateWindow (hwnd);
    return 1;
}
```

If you jumped ahead and executed this program, you would have discovered this produced the same results as the *hello world* program in Listing 1-1. Only a Windows message box is displayed with no window. What happened to the window?



The Message Loop

In MS-DOS, an application can instruct the operating system to display a graphics window. In an event-driven environment, such as Windows 95, the application and the operating system interact through messages. Your WinMain(), like most Windows applications, calls the function UpdateWindow() during initialization. The Windows 95 operating system responds to UpdateWindow() by sending your application a WM_PAINT message. Currently, your application has no mechanism for reading messages from the operating system nor does it have a message loop. A message loop can easily be added to your WinMain() by replacing your old code,

```
ShowWindow (hwnd, nCmdShow);
UpdateWindow (hwnd);
MessageBox(hwnd,"Hello World", "Spells of Fury", MB_OK);
return 1;
```

with code for a message loop, which looks like,

```
MSG msg;
ShowWindow (hwnd, nCmdShow);
UpdateWindow (hwnd);
MessageBox(hwnd,"Hello World", "Spells of Fury", MB_OK);
while (GetMessage (&msg, NULL, 0, 0))
```

continued on next page

continued from previous page

```
{
      TranslateMessage (&msg);
        DispatchMessage (&msg);
}
return msg.wParam;
```

Your code modifications to WinMain() include a while loop and a return value of *msg.wParam* instead of a 1. Windows 95 messages are 32-bit values, unlike Windows 3.1, which are 16-bit. A Windows application receives its messages in a MSG structure. Messaging will be discussed in detail in a section devoted to this topic later in this chapter. The concept to understand here is that the WinMain() function has a message loop to send and receive messages using the Win32 API calls DispatchMessage() and TranslateMessage().

Even with a message loop added to your WinMain(), your code still has one problem you must fix to display your window. There is no interface for the Windows 95 operating system to communicate with your application. A special function provided by the programmer, called the WndProc, is required for a Windows application to receive messages from the operating system. The WndProc is the topic of the next discussion. You will construct a minimal WndProc function to display your window.



The WndFunc() Function

As you have seen with your Example 2 code, a Windows application needs more than just a message loop to interface properly with the operating system. The window being displayed needs a special function for the Windows 95 operating system to send it messages. Remember when you set up your WNDCLASS struct, you initialized the *lpfnWndProc* field in the following manner:

```
wndclass.lpfnWndProc = NULL; // window function
```

You constructed a WNDCLASS and registered the class with no mechanism for the operating system to send messages to this window class. The *lpfnWndProc* field contains the address of the window callback function for this window class. Since you did not specify a function, the Windows 95 operating system has no way of sending messages to your window class to display it. You need to make the following changes to your WinMain() code:

```
wndclass.lpfnWndProc = WndFunc; // window function
```

The windows callback function can have any name you want to give it. In your case, you labeled it with a meaningful title, WndFunc(). Your windows callback function, WndFunc(), is the Windows 95 message handling function associated with the window registered in the WNDCLASS. The prototype for your WNDCLASS windows callback function is shown in Figure 1-4.

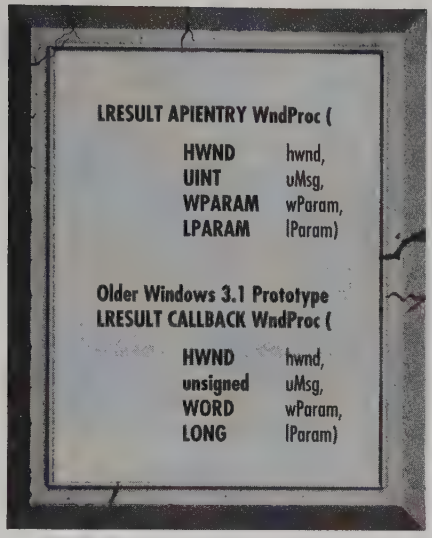


FIGURE 1-4
Designing a WNDCLASS WndProc function

The code you want to associate with this window's message handling is then provided by the application programmer in a function called WndFunc(). The title is arbitrary. You are using the name WndFunc() because this is the name you used in the WNDCLASS registry structure. Your WndFunc() window procedure is shown in Listing 1-3.

LISTING 1-3 Example 3: Using a WndProc for message handling

```
LRESULT APIENTRY WndFunc(HWND hwnd, UINT message, WPARAM wParam,LPARAM lParam)
{
        HDC         hdc;
        PAINTSTRUCT ps;

        switch (message)
        {
          case WM_PAINT:
            hdc = BeginPaint (hwnd, &ps);
            EndPaint (hwnd, &ps);
            return 0;

          case WM_DESTROY:
            PostQuitMessage (0);
            return 0;
        }
        return DefWindowProc (hwnd, message, wParam, lParam);
}
```

Your function WndFunc() is very simple for message handling. You handle the case for a WM_PAINT message, to paint the screen, and the case for the message WM_DESTROY, to stop the message loop and close the application.

Your WndFunc() is capable of handling hundreds of Windows 95 messages but you will only handle these two cases for right now. If you wanted to handle mouse or keyboard messages in your window you would add the code to this function.

You have also included the Windows API call DefWindowProc(), which is the default window procedure, for handling Windows messages your WndFunc() function doesn't process. This way, you can be assured that your application is capable of handling every message the Windows 95 operating system gives us. You'll notice that DefWindowProc() has the same calling arguments as a callback function.

Now, when you execute Example 3, your *hello world* message box is in focus, overlapping the window you worked so hard on getting to display. You can click OK on the message box to make it go away. When you do this, the application window's title bar changes from gray to blue, as shown in Figure 1-5. Windows notifies the user with the blue title bar, assuming a default Windows color scheme, that the current application window is in focus.

The window in focus has all the attributes of a typical Windows 95 window. Clicking on the system menu icon in the upper-left corner of the window pulls down the system menu to close the application. You can select to minimize, maximize, or close your application by clicking on the respective box located in the upper right of your window. These are all default settings for a Windows 95 application window.

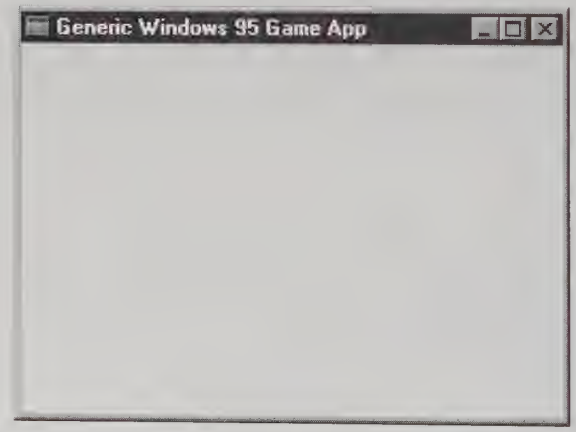


FIGURE 1-5
The Generic Windows 95 Game Application window



Windows 95 Programming Review

A Windows 95 application's start-up code calls the WinMain() function. The WinMain() function is responsible for all of the application's initialization. The WinMain() function registers a window class. Windows requires that a registered window class have a window class name and an address to the window callback that is capable of processing the posted or sent messages to the window.

A Windows application requires a windows callback function to process messages from the operating system. Messages to and from the application are handled by the message loop in WinMain(). With these simple concepts in mind, you can move forward and design an application specifically for the Windows 95 Game SDK.



A Generic Game Application

The code for Example 3 demonstrates how to create a simple Windows 95 application and display a window on the screen. You are going to add and modify this code to turn your application window into a game console style window. The code presented here will provide you a starting point for each time you want to develop a new application using the Microsoft Windows 95 Game SDK. It is designed for rapidly prototyping Windows 95 Direct X games so you don't have to keep going back and recoding a WinMain() and WndFunc() from scratch.



Forging a Game Console from Windows 95

The Windows 95 graphical user interface was designed for data entry. The concept behind the Windows interface is that all applications look and feel the same. This design feature would help in minimizing the time it takes for a user to learn how to use a new application. From an information systems and management perspective, Windows 95 is an ideal operating system for rolling out corporate-wide database retrieval applications. But for the game developer, this interface is highly unsuited for your applications. You want your user to

blast mutants or maneuver cute little characters over pits of spikes. Your graphical interface has to shout "Come and play!" and most of the time this means developing a unique and creative menuing system to catch your user's attention. The first thing you have to do to create a game console style application is to make all the look and feel of the Windows 95 GUI disappear. This will require some modification to your Example 3 program.



Initializing the Game Console Window

You will start your construction of the game console window just as you did with the Example 3 code. You define a window style you want displayed, using the WNDCLASS structure, and then you register the window. However, you won't call the Win32 API routine CreateWindow() to create a standard application window. You want a special window that lets the operating system know that no matter what happens this window is the topmost window until the application is terminated or minimized. You will want to use the Win32 API call CreateWindowEx() (creates a window with extended style), shown in Figure 1-6, to create your game console window.

When you initialize your game window, you are going to place all the WNDCLASS initializations of the window class registry, window creation, and other game initializations in a function called InitGameApp(). The function InitGameApp() will primarily be a cut-and-paste operation using the code from Example 3 to initialize a window in WinMain(). The exception here is that you will use the Win32 API routine CreateWindowEx() to create the game application window. The code for InitGameApp() is found in Listing 1-4.

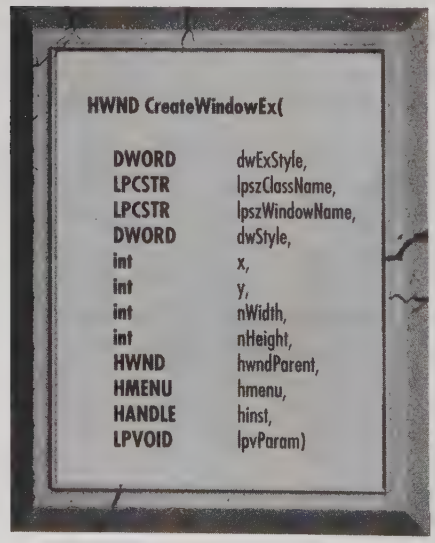


FIGURE 1-6
Win32 API call to create an extended window style

LISTING 1-4 Initializing the generic game application window

```
BOOL InitGameApp( HINSTANCE hThisInstance, int nCmdShow )
{
     HWND        hwnd;
     WNDCLASS    wndclass;
     char szAppName[] = "Generic Game SDK Window";

     wndclass.style         = CS_DBLCLKS;
     wndclass.lpfnWndProc   = WndFunc;  // window function
     wndclass.cbClsExtra    = 0;        // no extra count of bytes
     wndclass.cbWndExtra    = 0;        // no extra count of bytes
     wndclass.hInstance     = hThisInstance; // this instance
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW);
     wndclass.hbrBackground = GetStockObject (WHITE_BRUSH);
     wndclass.lpszMenuName  = NULL;
     wndclass.lpszClassName = szAppName;

     RegisterClass (&wndclass);

     hwnd = CreateWindowEx (
                    WS_EX_TOPMOST,         // window style
                    szAppName,             // window class name
                    "Generic Windows 95 Game App",// window caption
                    WS_VISIBLE | WS_POPUP, // window style
                    CW_USEDEFAULT,         // initial x position
                    CW_USEDEFAULT,         // initial y position
                    640,                   // initial x size
                    480,                   // initial y size
                    NULL,                  // parent window handle
                    NULL,                  // window menu handle
                    hThisInstance,         // program instance handle
                    NULL);                 // creation parameters

     if (!hwnd)
       return FALSE;

     ShowWindow (hwnd, nCmdShow);
     UpdateWindow (hwnd);
     SetFocus(hwnd);
     ShowCursor(FALSE);
     return TRUE;
}
```

All of the code presented here for the WNDCLASS and its registration is identical to Example 3. You use the function call CreateWindowEx() to create the game window. You set the *dwExStyle* parameter to WS_EX_TOPMOST, as was discussed earlier. You also have changed the parameters passed in for *dwStyle*. You logically OR the macro WS_VISIBLE with WS_POPUP. The *dwStyle* WS_POPUP makes your Windows style borders disappear. When you

execute the program you see nothing but a blank white screen. This is your game window. You may choose to use a different brush style instead of WHITE_BRUSH in the WNDCLASS window background field. For illustrative purposes, I chose white. The background set to BLACK_BRUSH sets the background to black. Using NULL_BRUSH, whatever is currently on the screen is displayed. If you execute the program using NULL_BRUSH from your project workspace, you'll only see your project workspace. It's confusing at first to tell if the application even worked. For this purpose, I select a WHITE_BRUSH.

Two other Windows API functions were added to your application initialization. These new functions are SetFocus() and ShowCursor(). You use SetFocus() to direct all keyboard input to your game application window. Whatever application had the keyboard focus previously has lost it. You also want to hide the mouse cursor. You do this by passing a Boolean value of FALSE to ShowCursor().



The Generic Game Application's WinMain()

Obviously, the cut-and-paste operation on WinMain() has left it in an altered state. You have also added some modifications to WinMain() as well during the code's metamorphosis. By calling InitGameApp(), WinMain() still initializes and registers the WNDCLASS and then creates it. Listing 1-5 shows the resulting WinMain() from all of your changes.

LISTING 1-5 The generic game application's WinMain()

```
int WINAPI WinMain (HINSTANCE hThisInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam, int nCmdShow)
{
     MSG msg;
     lpszCmdParam = lpszCmdParam;
     hPrevInstance = hPrevInstance;

     if (!InitGameApp(hThisInstance, nCmdShow))
       return FALSE;

     while(1)
     {
       if(PeekMessage(&msg, NULL,0,0,PM_NOREMOVE))
       {
         if(!GetMessage (&msg, NULL, 0, 0))
             return msg.wParam;
         TranslateMessage (&msg);
         DispatchMessage (&msg);
       }
       else
       {
```

```
            // do critical game stuff
        }
    }
    return msg.wParam;
}
```

A significant change is made in your message loop. Your message loop consists of an endless while() loop. Within the body of the loop you perform two primary operations: processing the game messages and executing the critical game code. Look at the flow of control in the while loop. When there are no messages to process, the program executes the critical game code. Your critical game code is the screen updating, game event handling, and so on. I use the term critical code in the concurrent programming sense. That is, program code that will execute with minimal amount of interruptions or halting for message processing, such as your game code.



The Generic Game Application's WinFunc()

When you created the window using CreateWindowEx(), you removed the system menus and title bar by specifying the window style WS_POPUP. You're going to have to provide your own hook to terminate the application without the use of a menu or a mouse for that matter. The code for the generic game application's WinFunc() is shown in Listing 1-6.

LISTING 1-6 The generic game application's WinFunc

```
LRESULT APIENTRY WndFunc(HWND hwnd, UINT message, WPARAM wParam,LPARAM
lParam)
{
       HDC         hdc;
       PAINTSTRUCT ps;

       switch (message)
       {
         case WM_PAINT:
           hdc = BeginPaint (hwnd, &ps);
           EndPaint (hwnd, &ps);
           return 0;
         case WM_KEYDOWN:
           switch(wParam)
           {
             case VK_ESCAPE:
                PostQuitMessage(0);
                return 0;
           }
           return 0;
         case WM_DESTROY:
```

continued on next page

continued from the previous page

```
            PostQuitMessage (0);
         return 0;
      }
      return DefWindowProc (hwnd, message, wParam, lParam);
}
```

The code is nearly identical to Example 3 except for the addition of handling a WM_KEYDOWN message. The generic game application will terminate when the user presses the Esc key. On PC games, the Esc key is usually used for invoking a game menu, such as the type used in *Doom* and *Heretic*. The generic game application is used for game prototyping. You may want to change the Esc key feature to a full-fledged bitmapped menu in your final application. For the examples in this book, the Esc key will terminate the application.



The Generic Game Application in Review

The generic game application serves the purpose of removing the Windows 95 style user interface to make way for a game console style interface. It also provides a starting point for developing any program using the Microsoft Windows 95 Game SDK libraries.

The initialization of the game display window has been moved from WinMain() into a function called InitGameApp(). InitGameApp() is responsible for registering the game display WNDCLASS as well as creating the game window. The function InitGameApp() is called by WinMain() for application initialization.

The WinMain() message loop is changed to provide the time slice for executing critical game code. When messaging is idle, the game code executes. The WinMain() function terminates when it receives a WM_KEYDOWN message with the value VK_ESCAPE.



Win32 Messaging

To introduce the concepts of a Windows 95 application, I breezed over the messaging process. The standard message loop used in Example 3 differed significantly from the message loop used by your game application. In this section, you'll examine the message loop you'll use for a generic game application as well as some important messaging concepts.

One of the design requirements for your generic game application is that you get messages from the message loop as quickly as possible to turn the processing time back over to the game application. In Example 3, the message loop waited for an event in the message queue. When there was no message, the application was idle. In your generic game application you cannot tolerate

any delay between waiting for messages. You're going to have to push the message loop. If the message queue has no message (it's empty), processor time will be handed over to the critical game process.



Processing Game Application Messages

When programming a game in MS-DOS you would make explicit calls, such as _kbhit(), to see if a key was pressed. The Windows architecture, on the other hand, doesn't provide a means to directly examine input. Rather, a Windows application waits to have input sent to it. When input, such as a keypress, occurs, the Windows 95 operating system passes the input to the destination application. Windows applications, such as the ones you've examined so far, all have windows callback functions that are called by the operating system whenever it has input for the application. The WndProc function processes the input and returns control to the Windows 95 operating system.

The WndProc function receives input in the form of messages. When the user moves a mouse, presses a key, or moves a joystick, the Windows 95 operating system sends a message signaling the input event. If the user maximizes or minimizes a window or changes the palette, the operating system sends a message to notify the application or other applications of the change. An application can also send an intraprocess message to one of its child windows. Applications can also use messages for interprocess communication between two applications.



Messages

Windows 95 uses a four member message structure, MSG, to communicate among its processes. The definition of the MSG structure is shown in Figure 1-7. The HWND field of the MSG structure identifies the destination window of the message. This field is used to determine which Windows callback function to call to process the message. The UINT numeric message field of the MSG structure is where the actual message is placed. Messages are defined in the WINDOWS.H header file. In your code you will use the macro name to identify a Windows message. These macros include WM_PAINT, WM_KEYDOWN, WM_MOUSEMOVE, and many others. The WM_ prefix identifies the macro as a Windows message.

When an application receives a message, it reads this numeric message field. The message in this field can be WM_PAINT, WM_MOUSEMOVE, or whatever. The application uses this field to determine how it will process the event. If the message field of the MSG structure contains the value WM_KEYDOWN, your generic game application will check to see if the message signaled that the VK_ESC key was pressed. How does your WinFunc()

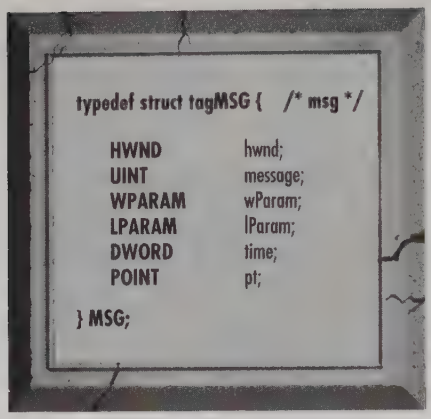


FIGURE 1-7
Windows 95 MSG structure

code know if a VK_ESC is pressed from the information in the MSG structure? A Windows message also contains data fields to provide additional information about the message. The fields WPARAM and LPARAM (see Figure 1-7) are used to hold message specific information. These fields can hold anything from data and packed data to addresses of data.

Processing the WM_KEYDOWN message as an example, your WinFunc() function examines the WPARAM field of the message for the WM_KEYDOWN message; this is the virtual key (VK) parameter. This is the key code that is returned from the key down event. The LPARAM field of the message is the key data field. The values found here include key repeat count, scan code, and key state.

In Chapter 4, you will add mouse functionality to your generic game application by processing WM_MOUSEMOVE events. You'll quickly look at this message just to see how the usage of the message data fields can differ. When a cursor move is detected, the operating system posts a WM_MOUSEMOVE message. The message sent to WinFunc() will contain information on the cursor's x and y coordinates on the screen. The WPARAM field of the MSG struct contains information about virtual keys specific to the mouse. For instance, was the left mouse button pressed by the user? The LPARAM field contains the x and y coordinates of the mouse (relative to the upper-left corner of the client area) at the moment the message was sent. You have two values, the x and y coordinates, packed in one field. To unpack these values from the LPARAM field, you will use the Windows macros LOWORD() and HIWORD(). These macros return the low word and high word of a long integer. To unpack the x and y values you would do something like this:

```
mouse_posx = LOWORD(lParam);
mouse_posy = HIWORD(lParam);
```

You will examine these macros further when you add mouse functionality to your generic game application. Just be aware that the data in the Windows message LPARAM and WPARAM fields are as unique as each message. You'll have to consult your compiler help files, MSDN CD-ROM, or other references to determine how a message presents its data in the MSG structure.

The last two members of the MSG structure are the *time* and *pt* fields. The *time* field contains the time the message was posted in milliseconds. The *pt* (point) field returns the screen coordinates of the mouse at the time the message is sent.



A Game Message Loop

When you first examined the message loop with Example 3, you established your messaging based on the return value of the Windows API call GetMessage(). Later, for your generic game application, you replaced the while-GetMessage loop with a simple endless loop. Your message loop for the generic game application looks like this:

```
while(1)
{
        if(PeekMessage(&msg, NULL,0,0,PM_NOREMOVE))
                // handle Windows message
        else
                // do critical game stuff
}
```

When the game application is started, the operating system creates an application queue for this task. The queue is used to store messages posted to your application. The queue has a default size of eight messages. Using the Win32 API call PeekMessage(), you can examine your application queue to determine if there are any messages to process. If there is a message, PeekMessage() will return a value of TRUE (see Figure 1-8). You pass the PM_NOREMOVE flag into the *wRemoveMsg* parameter of PeekMessage(). This specifies that you only want to examine the queue to see if a message is there and not to process it.

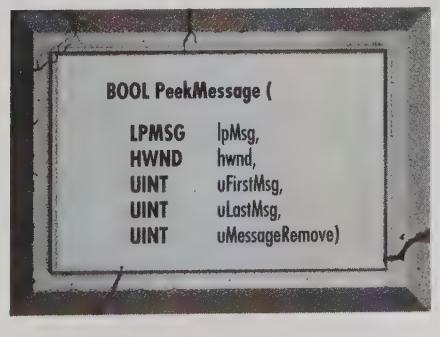


FIGURE 1-8
Checking the message queue

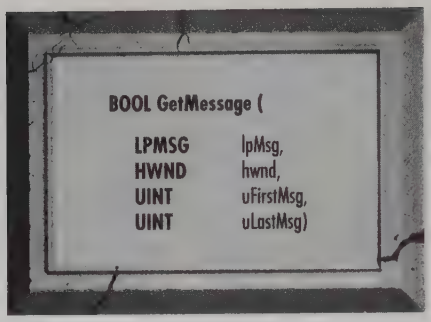


FIGURE 1-9
Retrieving a message from the queue

Why use PeekMessage() and not GetMessage()? By using PeekMessage() you are minimizing the amount of time your application is idle. GetMessage(), shown in Figure 1-9, waits for a message if no message is in the queue. The application is essentially put to sleep for a finite amount of time until a message is available. For database querying and office applications, this is acceptable. However, in your game application, you need to be conducting screen updates and processing background operations as quickly as possible. You don't have the luxury of having your application put to sleep until a message arrives in the queue. For this reason, you use PeekMessage() to check the queue. If there is a message waiting, the program calls GetMessage() to retrieve the message. Otherwise, process control is relinquished to your critical game code.



Message Handling

Now you'll examine how your generic game application processes a message from the application queue. You're going to fill in the blanks for your comment *// handle windows message* in the message loop code snippet presented previously. In this section of code, the Win32 API call PeekMessage() has returned a value of TRUE indicating a message is waiting in the queue. The program then calls its message processing code, which looks like this:

```
if(PeekMessage(&msg, NULL,0,0,PM_NOREMOVE))
{
        if(!GetMessage (&msg, NULL, 0, 0))
                return msg.wParam;
        TranslateMessage (&msg);
        DispatchMessage (&msg);
}
```

The Win32 API call GetMessage() is used to retrieve your message from the queue by copying the message to the application's MSG structure. GetMessage() returns a Boolean value based on the message it retrieves from the application queue. If the message retrieved from the queue is WM_QUIT (0x0012), GetMessage() returns the value of FALSE, terminating the message loop and WinMain(). All other messages return a value of TRUE.

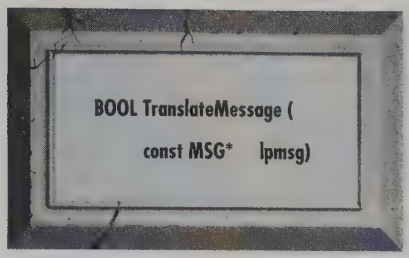


FIGURE 1-10
Translating virtual key messages to character messages

Once GetMessage() has placed the message into the MSG structure, two Win32 API functions are called, TranslateMessage() and DispatchMessage(). The function TranslateMessage() (see Figure 1-10) translates keystroke messages into character messages. If the message is a WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN, or WM_SYSKEYUP, TranslateMessage() places a WM_CHAR, WM_DEADCHAR, WM_SYSCHAR, or a WM_SYSDEADCHAR in the queue. The TranslateMessage() function takes virtual key messages (keystrokes) and translates them into character messages. The character messages are posted back into the application's queue to be read in by the next call to GetMessage().

The Win32 API call DispatchMessage(), shown in Figure 1-11, sends the message to the window procedure. For the generic game application, this is the WinFunc() function. The MSG struct contains the handle HWND to specify which Windows procedure is associated with this message. Recall that you registered your window procedure WinFunc() with the WNDCLASS structure. This is how DispatchMessage() knows which Windows procedure to use. Some programming examples may use the macro HWND_TOPMOST. In a multiple window environment, DispatchMessage() sends the message to all the top level windows. In the event the HWND parameter is NULL, DispatchMessage() does nothing with the message.

The function DispatchMessage() is responsible for dispatching messages to the appropriate window. For your generic game application, you have only one window the size of the display. However, it's conceivable that some Microsoft Game SDK applications may run in a bordered window on the Windows 95 desktop. These applications could have child windows like maps or some other interface to support the game.

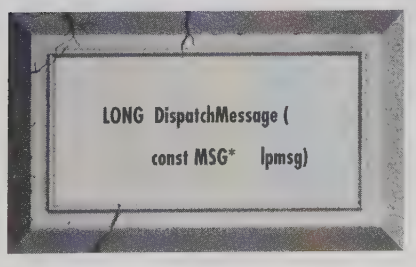


FIGURE 1-11
Dispatching a message to the Windows procedure

Game Messaging in Review

When a Windows application is executed, the operating system creates a message queue associated to the application. A message queue is capable of holding eight entries. The Win32 API function, PeekMessage(), is used to check the message queue for waiting messages. If a message is available, it is processed. Otherwise process control is relinquished to the critical game code.

The Dialog Box

A dialog box is a special window that allows the user to interact with the control of the program. The dialog box allows a user to select an item from a list or configure settings of a Windows application. When you are introduced to DirectPlay in Chapter 10, you will use dialog boxes to allow your user to choose a game communication service for multiplayer games. In this section, you'll examine the fundamentals of a dialog box.

A Very Simple Dialog Box

The dialog box presented in this example is a simple add-on to the generic game application. Most books I own on Windows programming actually make dialog boxes appear more difficult than they actually are. The examples always include radio buttons or other doodads to liven up the dialog box. In the presentation of dialog box, I feel less is actually better. Because dialog boxes are so easy, once you grab on to the concept, you can easily add the radio buttons and doodads yourself by just playing around with the code.

The DlgProc()

A dialog box is just a special kind of window that pops up over your application window. You learned from Example 3 that in order to display your window you needed a special window procedure (WndProc()) to process your messages from the operating system. To use a dialog in your code you must provide a dialog procedure (DlgProc()) for Windows 95 to send messages to your dialog box. A dialog box is nothing more than a special window. The function prototype for a DlgProc() is the same as the WndProc() (see Figure 1-12).

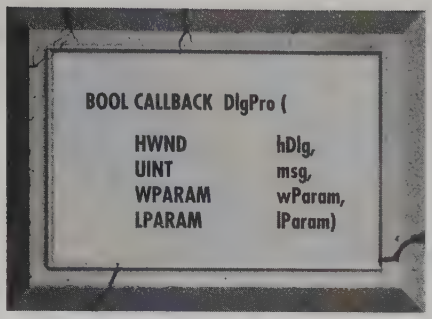


FIGURE 1-12
Dialog box procedure

As you can see from this function, the dialog box messages for your application are not sent to the game application's WinFunc() function. The dialog box handles its own messaging.

For this example, I used the Microsoft Developer Studio resource editor to create a dialog box for me. The dialog box for this example can be found in file DIALOG1.RC. I basically used all the default dialog box parameters for this exercise. The resource default labels are the same (see Figure 1-13). This will allow you to go in and play around with your resource editor without having to guess what I did to create this dialog box. The header file, RESOURCE.H, was generated for me by the resource editor. I did have to manually add this file to my source code to make the code compile with no errors.

The dialog callback code to handle the messages also has a similar appearance to the windows callback code. To keep the example as simple as possible, you only check for messages to determine if the OK button (IDOK) or the CANCEL button (IDCANCEL) was pressed by the user. If you're experimenting with the code on your own, be sure to set ShowCursor() in the generic game application's WinMain() to TRUE. Otherwise, the cursor will be hidden and selecting either of the two buttons will be a difficult task. The code for DialogFunc(), your dialog box message handling function, is shown in Listing 1-7.

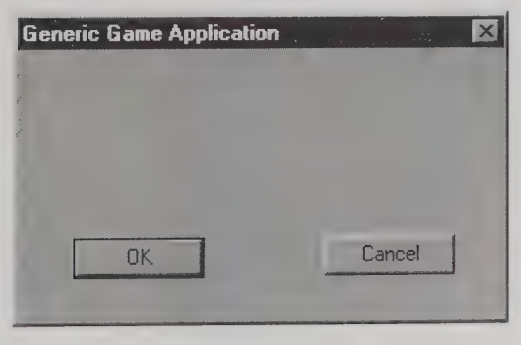


FIGURE 1-13
Dialog box used in this example

LISTING 1-7 Message handling function for the dialog box example

```
BOOL CALLBACK DialogFunc(HWND hDlg, UINT msg, WPARAM wParam, LPARAM lParam)
{
        switch(msg)
        {
          case WM_COMMAND:
            switch(LOWORD(wParam))
            {
              case IDOK:
              case IDCANCEL:
                EndDialog(hDlg, 0);
                return TRUE;
            } // switch(LOWORD)
        } // switch(msg)
        return FALSE;
}
```

Your dialog box, shown in Figure 1-13, has two controls, the OK and the CANCEL buttons. If the user selects either of these from your dialog box, a WM_COMMAND message will be placed in the message queue. Note that you extract which control was selected by using the LOWORD() macro to retrieve the data from the WPARAM field of the MSG structure. When the user selects either of these controls the dialog box will terminate using the API function EndDialog(), passing back a return code to the function DialogBox().



The Dialog Box Driver

You're going to need to add some code to your generic game application to execute your dialog box. You will use the Win32 API call DialogBox() (see Figure 1-14) to display the dialog box you created with the resource editor tools in the Microsoft Developer Studio. You can call the following code from WinMain() to activate your dialog box:

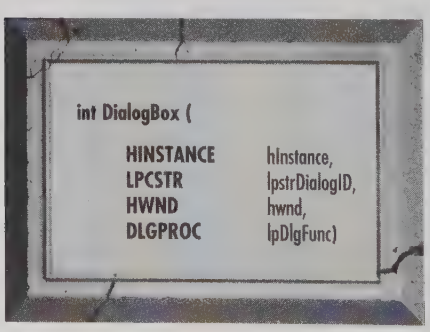


FIGURE 1-14
Activating a resource dialog box

```
int DialogDriver(HWND hwnd)
{
        return(DialogBox(NULL,(LPCSTR)IDD_DIALOG1,
                         hwnd, (DLGPROC)DialogFunc));
}
```

The routine calls the API function DialogBox() to active your dialog box. Windows sends the message WM_INITDIALOG to be processed by your dialog callback function. The dialog box appears, overlapping your game window immediately after WinMain() has finished its initialization. When the user selects any of the controls on the dialog, control of the program is returned to the calling application. In this case, the game window comes into focus.



The Dialog Box in Review

A dialog box is a popup window that overlaps the application window. A dialog box has a window callback procedure that is identical to WndProc for processing messages. Dialog boxes can be designed quickly with the resource editor in the Microsoft Developer Studio. The dialog boxes created there can be displayed using the Win32 API call, DialogBox().

The control buttons such as IDOK and IDCANCEL are located in the WM_COMMAND message and can be retrieved with the Windows LOWORD() macro. The dialog box terminates using the API call EndDialog().



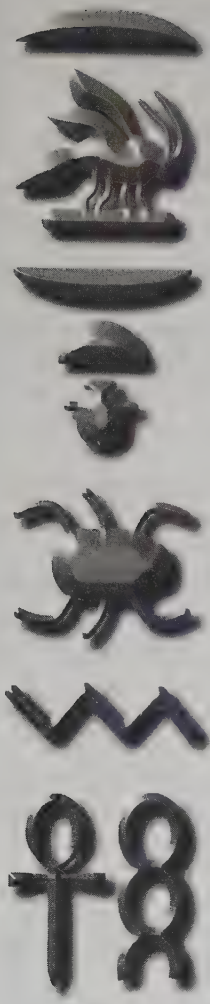
Conclusion

There was a lot of material in this chapter. It was kind of a high-speed primer for Windows 95 programming to get you up and coding quickly. If you're a DOS programmer hopping the fence to Windows 95, I recommend you add two books to your library, *Programming Windows* by Charles Petzold, and *Windows 95 Programming in C and C++* by Herbert Schildt. With these books, *Spells of Fury*, and an MSDN CD-ROM, you pretty much have all the resources you need at your fingertips. The Petzold book is on the MSDN CD-ROMs but it's not as easy to read in that format. Besides, there are times when a dog-eared page is easier to access than a CD-ROM. If you're a veteran Windows 3.1 programmer, this was kind of a refresher course. All of the functions you used in the 16-bit Windows API have pretty much migrated over to the 32-bit Win32 API.

In the next chapter, you're going to look at Device Independent Bitmaps (DIBs), which is the graphics format for Windows. But that's pretty much all you need to know at the moment about Windows 95 programming to use the Microsoft Game SDK DirectX libraries.

2

DIBs: TRANSLATING THE TABLETS



DIBs: TRANSLATING THE TABLETS

The Device-Independent Bitmap (DIB) is the graphics image format that is supported by Microsoft Windows. Other formats are used by the game industry but not supported by Windows. To migrate development over to the Windows 95 game SDK platform, an understanding of Device-Independent Bitmap is required. The topic of this chapter is working with the Microsoft Windows graphics format and how to program the formats with the same functionality you currently require for your game development needs.

EPISODE TRANSLATING THE TABLETS



It's now the second day of my stay at wherever it is I am. I suspect I am either in Nevada, at a site referred to by UFO enthusiasts as the government's Dreamland, or I am in New Mexico. Our location is subterranean with lots of tunnels and corridors. It's definitely not NORAD. I did a contract job there once and nothing here resembles or, I know it may seem a little weird, smells like the underground fortress of solitude used by our military in Colorado.

My current status: I'm a little grumpy. I didn't get to go for my morning surf. On top of that, Kala, myself, and a very young prodigy child, called 13, worked almost through the night. I don't think any of us were happy campers.

My young coworker approached me when I accidentally said a word I wasn't supposed to say out loud and in front of children. "Why are you so tense?" the boy, looking to be about nine years old, asked.

"I'm not tense," I replied. "Just a little put out that I didn't get to ride the waves this morning. I usually surf in the morning, near my home in Santa Cruz.

You ever heard of the place?" I figured I could trick the kid into disclosing my present location to me.

"Yeah, I heard about it this morning on the news as a matter of fact. Some guy was eaten by a great white shark out there yesterday. They only found a piece of surfboard that washed ashore," 13 answered.

I didn't find my location, but I did make the startling discovery that I didn't exist anymore. At least not publicly or electronically. Apparently my host had seen to that. My thoughts now focused on what my friend Dr. Goldmen may have stumbled across. Did they get rid of her and bring me in to finish their dirty work?

"So, 13, what does that number stand for. Is it like a 'Get Smart' agent 86 thing? Do you hide in vending machines or cigarette ashcans to hand off secrets to other spies?"

"What's 'Get Smart'? Who's 86? Did they make one that lives to be 86 now?" the boy replied. He never diverted his attention away from the computer terminal. It was inhuman. Without even thinking about what he was doing, he was writing a computer program at a rate of about 20 lines per minute.

During my stimulating conversation with 13, Kala walked up and inconspicuously slipped a small notebook into my back pocket.

"There are cameras and microphones everywhere," she whispered trying not to draw any attention. "The young boy is a synthetic life form. He is only designed to live 13 years. Our employer fears his super intellect, so he insisted on a finite life span."

"13 is a product of Project GNOME?" I responded quietly. Kala jotted down some notes regarding the tablet we were studying.

"Yes, he is. But not the genetic research project you and I read about in scientific journals. 13 is comprised of non-terrestrial DNA. There are a lot of weird things here. Just pretend like nothing phases you and you'll do fine. Our friend Carol, Dr. Goldmen, has a high level of confidence in you. I need your help Bary. Carol and I both need your help," Kala requested.

"Kala, I don't know you. And based on current events, I can't trust you. Not yet, anyway. I do trust Carol, though. I want to see her. Where is she?" I replied.

Kala gave 13 a confidential head gesture to head me to Dr. Goldmen. Apparently, the two of them had a plan already worked out. What I didn't anticipate was what they had to show me. Dr. Goldmen was interfaced to the ancient artifact with an advanced virtual reality suit, the likes of which I had never seen before. A cut was made in the suit to attach an IV and some heart monitors.

"We found her like this three days ago," 13 divulged to me. "She has had some sort of seizure. I added the intravenous and life support equipment at Kala's request. But it is my sound judgment that your friend is brain dead."

The wires and connectors to the VR suit led into the room next to ours. Apparently, Carol had entered the artifact using virtual reality technology.

"Oh boy..." I muttered out loud.



"What's wrong?" Kala asked with a concerned tone.

"I'm not positive, but the zero contingency virus had one minor design flaw. It needed a sentient driver program to control it. Our artificial intelligence programs we were working on were nowhere near being sentient. But Carol found a way. She always did. Virtual Reality. ...It never even dawned on us. What is this thing anyway?" I asked.

"It's VR AssaultWare. I designed it," 13 stated factually and not in a bragging tone. "Carol didn't wait for me, though. This is one of my prototypes. She didn't wait for me to assist her. That's what happened," the boy mumbled defensively.

"No, 13. She did this on her own. None of it's your fault. She downloaded her mind into the artifact; she's the control program for the virus she inflicted Agamemnon with. She figured out how the old dude did it. Which means, hopefully, she knows how to get out too. 13, you said this is one of your prototypes?"

"Yes, I have three more," 13 answered.

"Get one ready. I'm going in after Carol. Then we're all getting out of here."

"Bary, the notebook I put in your pocket contains information Carol wanted you to see before you attempted anything," Kala said as she adjusted the glasses on her face.

"Thanks Kala. When the Grinch is finished with his assault on Whoville or stapling the antlers on his dog's head, he's going to come looking for you. I need you to keep him busy until 13 and I are finished. Then we all bail out of here together!"



The Device-Independent Bitmap

A majority of the commercial MS-DOS games on the market use the Electronic Arts Interleaved Bitmap (ILBM) and PCX graphics formats. This is primarily because the Commodore Amiga, using Deluxe Paint, was the game industry's premier platform for developing animation cells and art for video games. Today, the Amiga has pretty much disappeared from the industry, but the Deluxe Paint application once used on the Amigas is still a popular tool for artists and is now being used on PCs.

There is a subtle discrepancy when you migrate over to the Windows 95 Game SDK platform. Microsoft Windows only understands the bitmap (BMP) graphics file format. Most PC game art is designed for a screen resolution of 320 x 200 pixels (VGA Mode 13h) or 320 x 240 pixels (Mode X, discussed in the next chapter), where the Windows 95 Game SDK resolution is 640 x 480 pixels. The Windows API supplies its own routines for creating, reading, and blitting its own bitmap architecture, called a Device-Independent Bitmap (DIB).



The DIB Architecture

A DIB is stored in a Windows .BMP file with its image bits and color table. Both of these components must be retrieved from the file to properly display a DIB. The Windows .BMP file structure is shown in Figure 2-1. For those of you familiar with PC game file formats, such as IFF, LBM, and PBM, you'll notice that the image isn't stored in chunks.

The DIB stored in a Windows .BMP file is composed of three data structures. These include the BITMAPFILEHEADER, the BITMAPINFOHEADER, and a buffer for the image bits. The first structure in the file is the BITMAPFILEHEADER. The first field in this structure, *bfType*, is the first word in the file. It contains the ASCII value "BM" (0x4D42) for bitmap. This identifies that the file is a .BMP file. Another field within the first couple of bytes of the file is *bfOffBits*. This is the offset in bytes from the start of the file to the image bits stored in the file. Figure 2-2 shows how the offset field is used to locate the image in the .BMP file. The BITMAPFILEHEADER is primarily used to identify the file type and the location of the image bits in the file.

Just after the BITMAPFILEHEADER, in the .BMP file, is the BITMAPINFOHEADER structure which contains specific information and attributes about how the DIB image is to be displayed. These fields include the DIB image size, image width, image height, compression, and resolution, among other values. Just after the BITMAPFILEHEADER are the DIB 32-bit color table entries called the RGBQUAD. These structures with a DIB storage example are shown in Figure 2-2.

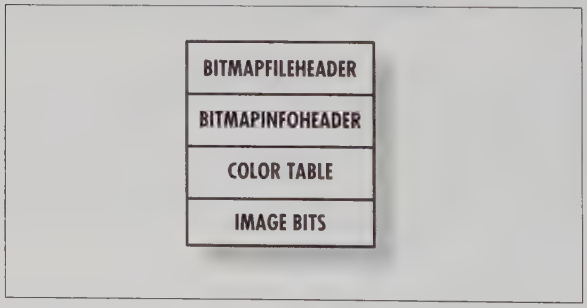


FIGURE 2-1
Windows .BMP file structure

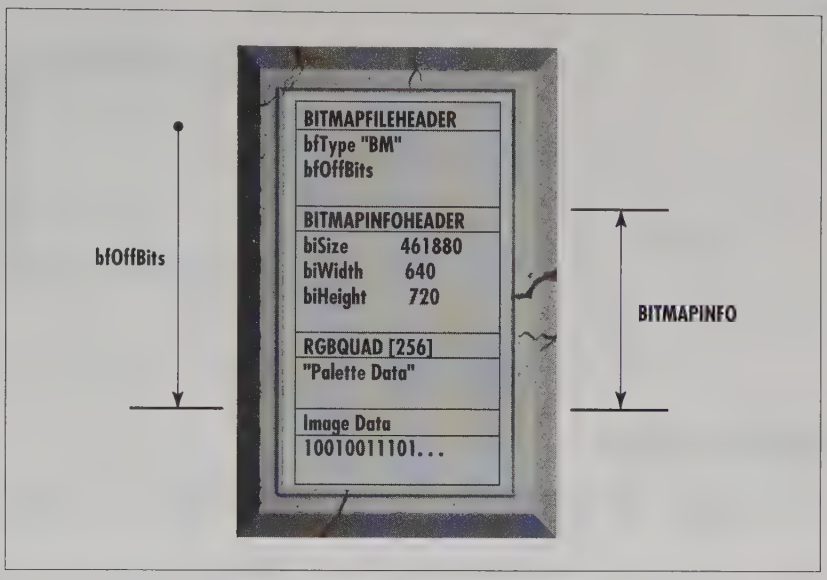


FIGURE 2-2
Example of a DIB stored in Windows .BMP file

DIBs and System Memory

When a DIB is read in from a .BMP file it is typically stored in two memory buffers, as shown in Figure 2-3. The first buffer is the BITMAPINFOHEADER from the .BMP file. The BITMAPINFOHEADER, as you recall, contains the attributes about how the DIB is to be displayed. The BITMAPINFOHEADER also contains the color table entries for the image. A second buffer is necessary to hold the image bits, which are stored external to the BITMAPINFOHEADER data in the .BMP file.

When your application reads in a DIB from a .BMP file it will have to create and maintain these two separate buffers. Figure 2-4 shows a Windows 95 application with a DIB stored in system memory. The application has a buffer for the image bits and a buffer for the BITMAPINFOHEADER data. This application code has read a DIB in as a resource and loaded it into memory.

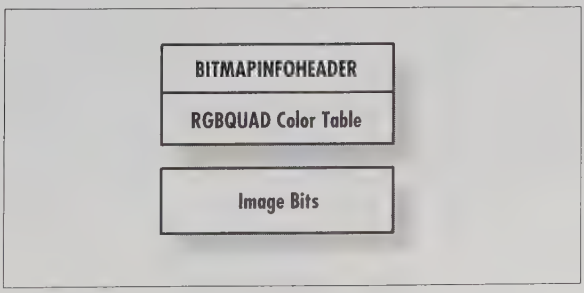


FIGURE 2-3
DIB comprised of two buffers in memory

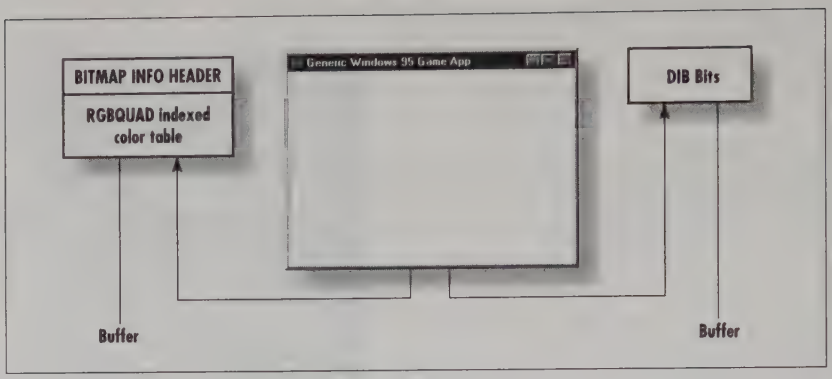


FIGURE 2-4
Application with a device independent bitmap

CDib: Designing a DIB Class

The Microsoft Windows API provides you with some routines to load DIBs as a resource and blit them to your application's window device context. But there really aren't too many utilities to return specific information about a pixel at a specific coordinate, or its color value and so on. For game development, programmers need access to a wealth of information about the images they wish to display. When you start using DirectDraw objects in Chapter 3, it will be necessary to break down a DIB into constituents that are readable by DirectDraw. One such operation is handing your DIB's RGBQUAD palette over to the DirectDrawSurface object in the form of palette entries and not a DIB palette as a whole.

You're going to need to fill in the blanks with some utilities that the Windows API didn't provide you with. To do this you're going to use the power of object-oriented programming (OOP) to create an interface that will manipulate and retrieve DIB information. For this discussion you don't have to be an OOP guru; the design will be simple. Along the way you'll pick up a few object-oriented design buzz words to impress your manager with. I have chosen not to use Microsoft Foundation Classes (MFC) for this interface. This is because I want some degree of flexibility if I ever have to port my C++ code over to another environment. It's common for popular game titles to be redistributed across multiple platforms. *DOOM*, for instance, is available for the PC, Macintosh, Next, Sun, SGI, Sega, Jaguar, and Super Nintendo. MFC is Microsoft Windows-specific. If the Microsoft Game SDK should migrate to a game console environment, such as the Sega Saturn or Sony PlayStation, MFC hooks probably won't be available.



CDib Class Declaration

The C++ DIB interface, which we'll call the CDib class, will provide the functionality to read a DIB from a.BMP file, retrieve its RGBQUAD color table, and maintain a pointer to its bits. The CDib class declaration looks like this:

```
class CDib
{
  protected:
    LPBITMAPINFO    lpbinfo;     // Pointer to BITMAPINFO struct
    BYTE*           lpBits;      // Pointer to the bits
    LPRGBQUAD       lpRGB;       //  Pointer to the RGB table

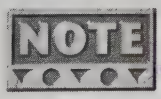
  public:
    CDib();
    ~CDib();
};
```

The class consists of three data members, *lpbinfo*, *lpBits*, and *lpRGB*. The *lp* preceding the data member names is a Windows coding convention to indicate a 32-bit pointer (*lp* for long pointer). A data *member* is a variable that belongs to a class in C++. All three data members in the CDib class are *protected*. This means that no other function or class that is not a *friend*, or member, of or derived from CDib may modify any of the data members.



CDib Class Constructor and Destructor

The two member functions under the *public* declaration are the class *constructor*, CDib() and the class *destructor* ~CDib. The tilde "~" in C++ identifies the destructor member function. The destructor is called to deallocate or de-initialize a C++ object. The constructor is called to initialize a C++ object.



Constructors and destructors in C++ do not have a return type.

In C++, you put your class declaration in a header file. A popular file naming convention is to give header files with class declarations an .hpp file extension. Other header files that don't have class declarations use the standard .h file extension. Your class declaration for CDib will be in the file cdib.hpp. The member functions for the class can be found in cdib.cpp. With these conventions in mind, you can start your design with the class constructor. The class

cDib constructor is declared in the header file and the code for the constructor, located in the source file, looks like this:

```
CDib::CDib()
{
  lpbinfo = NULL;
  lpBits = NULL;
  lpRGB = NULL;
}
```

For those of you fluent in C++, you may be more comfortable with the constructor in this form:

```
CDib::CDib():lpbinfo(NULL),lpbits(NULL), lpRGB(NULL)
{
}
```



The "CDib::" is the class scope. All member functions in the .cpp file must be preceded with the class scope operator to be identified by the compiler as a member function to the class.

The constructor merely initializes the data members, which are pointers, to NULL. Likewise, the class destructor ~CDib() de-initializes the object.

```
CDib::~CDib()
{
  if (lpbinfo != NULL)
    GlobalFreePtr(lpbinfo);
  if (lpBits != NULL)
    GlobalFreePtr(lpBits);
  if (lpRGB != NULL)
    GlobalFreePtr(lpRGB);
}
```

The code in the destructor is common to other member functions too. When a memory allocation fails, you need to clean up any pointers that may be initialized before exiting the code. These three conditional statements were popping up regularly as I developed CDib so I decided to convert this code into a member function called FreeCDib().

```
CDib::FreeCDib()
{
  if (lpbinfo != NULL)
    GlobalFreePtr(lpbinfo);
  if (lpBits != NULL)
    GlobalFreePtr(lpBits);
  if (lpRGB != NULL)
    GlobalFreePtr(lpRGB);
}
```

The destructor code now looks like this:

```
CDib::~CDib()
{
  FreeCDib();
}
```



In Win32 you're using GlobalAllocPtr() and GlobalFreePtr() instead of the C++ new and delete operators. The difference is that memory is being allocated from the system heap rather than the program's free store.

Now that you have the code to construct and destroy your CDib objects, add some functionality to it. When the constructor is called you initialize *lpbinfo, lpbits, lpRGB* pointers to NULL. Now take a look at how DIB data structures are organized in memory.



Forging a DIB with NewDIBObject()

The most fundamental task of class CDib is to create a DIB object. Most source code on the Internet and CD-ROMs have made this simple action look rather nasty. I have broken down the operations necessary to create a DIB object into four steps, as shown in Figure 2-5. The most important concept to understand is that a DIB, in system memory, is actually two buffers (see Figure 2-4). One buffer is the Windows BITMAPINFO data structure, which contains both the DIB's BITMAPINFOHEADER and the DIB's RGBQUAD color table. The second buffer required is the memory for the image bits.

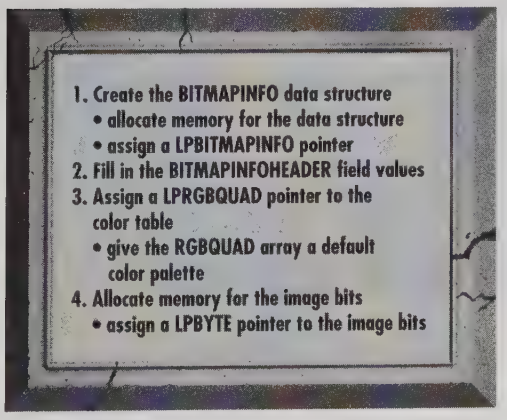


FIGURE 2-5
Steps to forging a DIB

Step 1. Creating the Windows BITMAPINFO Structure

The Windows DIB BITMAPINFO structure contains both the BITMAPINFOHEADER structure and the RGBQUAD color table (see Figure 2-6). The BITMAPINFOHEADER structure holds the information describing the DIB width, height, compression type, and how many colors are in the palette. The first step you need to take in creating the DIB is to allocate memory for the BITMAPINFO data structure. The amount of memory required for the data structure is the size of the BITMAPHEADER, which is 40 bytes, added to the size of the color table. The code for calculating the BITMAPINFO size for a color table that can hold up to 256 colors, looks like this:

```
int binfoSize = (sizeof(BITMAPINFOHEADER) + 256*(sizeof(RGBQUAD));
```

The variable *binfoSize* is now initialized to the size of memory you need to allocate for the BITMAPINFO structure (see Figure 2-7). Remember you will be requesting memory from the Windows heap and not the C++ free store. The class CDib data member *lpbinfo* will reference the pointer if the allocation is successful. The variable *binfosize* represents the size in bytes your BITMAPINFO structure requires. You can use this value to allocate the memory and assign a pointer to it. This code looks like this:

```
lpbinfo = (BITMAPINFO*)GlobalAllocPtr(GMEM_MOVEABLE, binfoSize);
```

Remember to always check and see that the allocation was successful. You would be surprised at how many professional programmers forget to do this. The code can be as simple as the following conditional statement:

```
if (!lpbinfo)
       return FALSE;
```

This completes the steps for creating a BITMAPINFO structure. Now you have to initialize the fields in the BITMAPINFOHEADER structure and fill in the RGBQUAD color table that is in the BITMAPINFO structure.

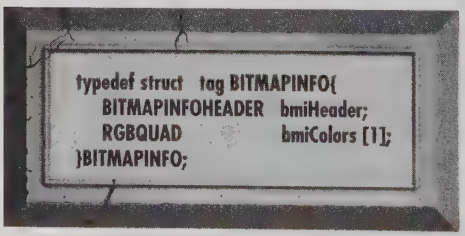


FIGURE 2-6
TYPEDEF of the BITMAPINFO structure



FIGURE 2-7
Memory required for BITMAPINFO



Step 2. Filling in the DIB's BITMAPINFOHEADER

You now have a default BITMAPINFO data structure in memory that contains no useful values. You need to fill in these attributes and give your DIB bitmap some dimensions. This is accomplished by properly filling in the BITMAPINFOHEADER fields (see Figure 2-8). To fill in the fields, you need to reference the fields. The BITMAPINFOHEADER is a structure inside the BITMAPINFO data structure. You can assign a BITMAPINFOHEADER pointer by typecasting the data member *lpbinfo* as follows:

```
BITMAPINFOHEADER* lpbi = (BITMAPINFOHEADER*)lpbinfo;
```

Using the BITMAPINFOHEADER pointer, *lpbi*, you can now assign values to the BITMAPINFOHEADER. The code looks like this:

```
BOOL CDib::SetBitmapHeader(int biWidth, int biHeight)
{
  if (!lpbinfo)
    return FALSE;
  memset(lpbi, 0, sizeof(BITMAPINFOHEADER));
  lpbi->biSize            = sizeof(BITMAPINFOHEADER);
  lpbi->biWidth           = biWidth;
  lpbi->biHeight          = biHeight;
  lpbi->biPlanes          = 1;
  lpbi->biBitCount        = 8;
  lpbi->biCompression     = BI_RGB;
  lpbi->biSizeImage       = 0;
  lpbi->biXPelsPerMeter   = 0;
  lpbi->biYPelsPerMeter   = 0;
  lpbi->biClrUsed         = 0;
  lpbi->biClrImportant    = 0;
  return TRUE;
}
```

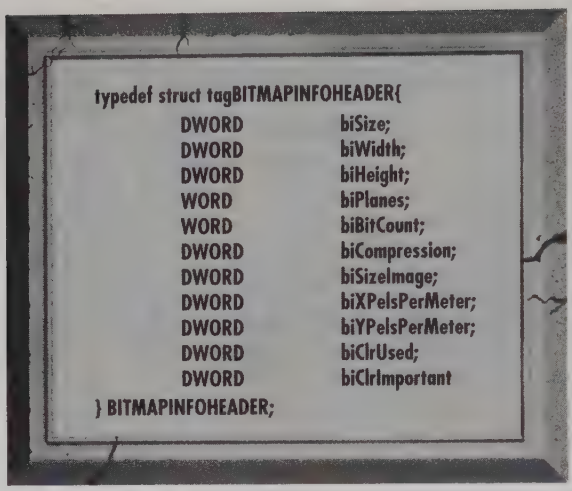


FIGURE 2-8
TYPEDEF of the BITMAPINFOHEADER structure

The BITMAPINFOHEADER values for *lpbi->biWidth* and *lpbi->biHeight* are passed into the function. The arguments passed in are *biWidth* and *biHeight*. This should help solve some ambiguity. The field *biBitCount* tells you the resolution of the image in bits per pixel (bpp). In this case, you'll be using 8 bpp bitmaps. If you're feeling a little adventurous, play around and try values of 16 and 24. Different compression methods are available, such as run-length encoding (RLE) and the option for no compression. You select no compression in this field with the constant BI_RGB (BI_RGB = 0). The field *biPlanes* describes how the bytes are arranged in the image. Some storage schemes use planes of memory. You are going to use a linear memory scheme. Linear memory is selected by setting this field to one for one continuous memory plane. The strange values *biXPelsPerMeter* and *biYPelsPerMeter* are the number of pixels per meter. You aren't particularly interested in these as a game developer. However, if you were trying to write some code to interface to a laser printer, then these fields would become more useful. You have seen in programs like *Photoshop* where the image ratio is measured in pixels per inch. This is the same idea, only it's metric. The last two fields tell how many colors are used. The *biClr* field holds the value of how many colors are in the DIB. A typical value for this field is 256 colors.

Remember, every time you request memory you must make sure the memory is allocated to you. The *if* statement precluding filling in the values for *lpbinfo* checks to see if *lpbinfo* is a valid pointer. The code following the memory check initializes your BITMAPINFOHEADER referenced by *lpbinfo*. There may be several different occasions where you would want to initialize a BITMAPINFOHEADER, so moving this chunk of code into a separate member function would be useful. The function definition looks like this:

```
BOOL SetBitmapHeader( int biWidth, int biHeight);
```

Second, pulling this entire block of code out of NewDIBObject() and replacing it with a single line call to a function made the code more readable. That's it for initializing the BITMAPINFOHEADER.



Step 3. Creating a Default RGBQUAD Color Table

The RGBQUAD color table is sitting in the BITMAPINFO data structure. Its values are assigned to whatever garbage happened to be in the memory at the time the BITMAPINFO structure was allocated. You need to assign a pointer to this array and initialize the RGB values in it.

If your data member *lpbinfo* is non-NULL, you can easily return a pointer to the RGBQUAD color table. This code can be placed in the header file since you're only returning a pointer. The code looks like this:

```
class CDib {
  protected:
    LPRGBQUAD lprgb;
        :
        :
  public:
    CDib();
    RGBQUAD* GetRGBQUADPtr()
    {
      return(LPRGBQUAD)(((BYTE*)(lpbinfo))+sizeof(BITMAPINFOHEADER));
    }
    ~CDib();
};
```

This member function is handy because it allows functions outside the class to read the color table, which is a protected data member. When you pass an RGBQUAD palette to a DirectDrawPalette object, this is the function you'll use. An important aspect of the *LPRGBQUAD* data member is that it references the inside of the BITMAPINFO structure at an offset of BITMAPINFOHEADER size. Remember, the color table is in the BITMAPINFO structure just after the BITMAPINFOHEADER structure.

Now that you can access the color table, it's time to fill it in. You can write a simple loop to initialize all 256 colors:

```
for (int i=0; i<256; i++)
{
  lprgb->rgbBlue = lprgbGreen = lprgbRed = (BYTE) i;
  lprgb->rgbReserved = 0;
  lprgb++;
}
```

The palette you just initialized won't be spectacular. It's actually a 256-color gray scale palette. This is adequate for new DIB objects. After all, your objective here is to just create an empty DIB and draw it on the screen.



Step 4. Allocating the Image Bits Memory

Well, enough about the BITMAPINFO memory. Now take a look at the image bits memory, which is the second buffer required for constructing a DIB. You must first know how much memory you need in order to allocate this buffer on the system heap. A first notion is to say this:

```
int lpbitsSize = biWidth * biHeight;
```

This is wrong. The DIB image bits are DWORD aligned. This means that the width of the scanlines are always divisible by four. Calculating the amount of memory required for the bits now looks like this:

```
int lpbitsSize = ((biWidth+3) & ~3) * biHeight;
```

This will determine how much memory is required for the DWORD aligned buffer. The allocation of memory is still done in the normal fashion:

```
lpBits = (BYTE*)GlobalAllocPtr(GMEM_MOVEABLE, lpbitsSize);
```

And of course, don't forget to check to see if you got the memory you requested:

```
if (!lpBits)
{
  FreeCDib();
  return FALSE;
}
```

If the allocation was successful, you now have a buffer to write to. The simplest method to write bits is to use *memset()* and fill the image bits RGB color table to RGB(0,0,0), which is black.

```
memset(lpBits, 0, lpbitsSize);
```

Now you have an image in memory. It's all filled with zeroes, but it's still an image with 8 bpp with a size of 640 x 480. These are the values you brute-forced, using your constructor and the call to NewDibObject(). It's not a good coding practice to always allocate a 640 x 480 DIB; it's purely for illustrative purposes. After a small discussion on drawing to a device context, you'll be ready to display your monochromatic work of art. For those of you who are into surrealistic art, or torture, you can pop open a bottle of wine, put out some cheese and a pate, turn on your old Beatles records, and have your own little art exhibit in front of your computer.



Drawing the CDib Object Bits to the Game Screen

Drawing your bits to the Generic Game Application screen requires a simple call to the Windows 95 API function StretchDIBits(). Since you are encapsulating DIB functionality into your cDib class, it would be nice to provide a member function that draws the DIB object to a device context (DC). To draw your DIB to a DC you will use the CDib member function DrawBits(). The first parameter in member function DrawBits() is a handle to a DC. The DC is a data structure owned by the Graphic Device Interface (GDI). The DC is attached to the screen device driver. When you draw the CDib object image bits you will be drawing them to the DC. The DC knows how to draw on the device, namely the screen in this case.

The second and third parameters passed to DrawBits() are the coordinates where you want the API function StretchDIBits() to start drawing. Passing values of zeroes for these two fields results in having the DIB object start its drawing in the upper left corner of the screen.

```
void CDib::DrawBits(HDC hDC, int x, int y)
{
       ::StretchDIBits(hDC,
       x,                                  // Destination x
       y,                                  // Destination y
       lpbinfo->bmiHeader.biWidth,         // Destination width
       lpbinfo->bmiHeader.biHeight,        // Destination height
       0,                                  // Source x
       0,                                  // Source y
       lpbinfo->bmiHeader.biWidth,         // Source width
       lpbinfo->bmiHeader.biHeight,        // Source height
       lpBits,                             // Pointer to bits
       lpbinfo,                            // BITMAPINFO
       DIB_RGB_COLORS,                     // Options
       SRCCOPY);                           // Raster operator code (ROP)
}
```

The actual drawing to the DC takes place when the GetMessage() function retrieves a WM_PAINT message. The code in the Generic Game Application would look like this to handle the WM_PAINT message and draw the CDib object bits:

```
LRESULT CALLBACK WndFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM
➥lParam)
{
   HDC     hdc;
   PAINTSTRUCT ps;
   switch (message)
```

```
    {
      case WM_PAINT:
      {
        hdc = BeginPaint (hwnd, &ps);
        pDib->DrawBits(hdc,0,0);
        EndPaint (hwnd, &ps);
        break;
      }
        :
    }

}
```



Creating a DIB in Review

Creating a DIB requires allocating memory for a BITMAPINFO structure. The BITMAPINFO structure contains a BITMAPINFOHEADER structure and an RGBQUAD color table. The BITMAPINFOHEADER fields can be accessed by typecasting the BITMAPINFO pointer to (BITMAPINFOHEADER*) and assigning a pointer to it.

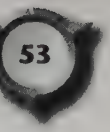
The RGBQUAD color table is located in the BITMAPINFO structure just after the BITMAPINFOHEADER. The color table can be referenced by assigning an LPRGB pointer to the BITMAPINFO structure at an offset of the size in bytes of the BITMAPINFOHEADER structure.

The image bits of a DIB are DWORD-aligned. The scanline widths are always divisible by four. The image bits of the DIB are in memory separate from the BITMAPINFO structure.

LISTING 2-1 Complete listing for the CDib class header file

```
#ifndef __CDIB__HPP
#define __CDIB__HPP

typedef enum enum_CDIB_States {
  CDIB_OK,
  FAIL_DIB_OPEN,
  FAIL_FHDR_OPEN,
  FAIL_FHDR_READ,
  FAIL_BFTYPE_INVALID,
  FAIL_IHDR_READ,
  FAIL_FORMAT_INVALID,
  FAIL_CORE_READ,
  FAIL_LPBI_ALLOC,
  FAIL_CLR_READ,
  FAIL_RGBQD_READ,
  FAIL_BITS_ALLOC,
  FAIL_BITS_READ
} CDibState;
```




```
////////////////////////////////////
// CDib Class Declaration
////////////////////////////////////
class CDib
{

  protected:
    LPBITMAPINFO    pbinfo;       // Pointer to BITMAPINFO struct
    BYTE*           lpBits;       // Pointer to the bits
    LPRGBQUAD       lprgb;        // Pointer to RGBQUAD
    CDibState       state ;       // current state of DIB

  public:
    RECT                bounds;
    CDib();
    ~CDib();

    BITMAPINFO* GetBitmapInfoPtr() {return lpbinfo;} // Pointer to bitmap
    ➥info
    BYTE*     GetBitsPtr() {return lpBits;}          // Pointer to the bits
    LPRGBQUAD GetLPRGBPtr(){return (LPRGBQUAD)(((BYTE *)(lpbinfo))
              + sizeof(BITMAPINFOHEADER));}          // Pointer to color
              ➥table
    BOOL     NewDIBObject(int width, int height);
    void      DrawBits(HDC hDC, int x, int y);
    void      DrawBits(HDC hDC, RECT *r);
    BOOL      SetBitmapHeader(int biWidth, int biHeight);
    BOOL      SetDefaultPalette();
    void      FreeCDib();
    BOOL      ReadFile(LPSTR lpszFilename);
    void      SetCDibState( CDibState estate );

    HPALETTE  CreateLogicalPalette();
    BOOL     CopyDIBBits( const CDib& srcDIB, RECT srcRect, RECT dstRect);
    BOOL     CopyLRGBQUAD( const CDib& srcDIB);
    UINT     GetBITMAPINFOSize()
             {
               return((UINT)
                    (sizeof(BITMAPINFOHEADER) + 256 * sizeof(RGBQUAD)));
             }
    BOOL     CreateBITMAPINFO();
    BOOL     CopyBITMAPINFO( const CDib& srcDIB );
    void     SetBounds(int left, int top, int right, int bottom);
    void     SetBounds();
    UINT     GetCDibBiHeight();
    UINT     GetCDibBiWidth();
    BOOL     CopyTransDIBBits( const CDib& srcDIB,
                   RECT srcRect, RECT dstRect, BYTE transByte);
    BYTE     GetPixelByte( UINT x, UINT y);
    COLORREF  GetPixelRGB( UINT x, UINT y);
};
```



CDib Member Functions for Reading a .BMP File

The Programmers Proverb

A long time ago someone wrote the very first computer program. And every programmer has cut, pasted, and modified this code ever since. —Anonymous

One of the most hideous aspects, I feel, of graphics programming is learning to work with a new file format. As a DOS programmer, I was more than pleased with the libraries of source code I had for Deluxe Paint LBM and PCX files. I could simply doodle with a paint program, save the format to my specifications, cut, paste, and compile some code, and voila! I was animating.

Then I found out I had to develop new libraries for Windows 95 to read DIBs. I searched the bookstores, the Net, and CD-ROMs for source code. The programmer's proverb was holding its ground here. Out of the mountains of code I obtained, I think I narrowed down about five source listings, which I believe to be some of the original programs for reading a DIB. One listing is difficult to read because it is in a now defunct ancient Tibetan language. As each source listing became another cut away from the original program, the more convoluted the code became. What I finally ended up doing was looking at a few source listings from a higher level, just glancing at the code, and trying to piece together what it was doing. The actual brainstorm came when I figured out that reading a DIB can't be much different from creating a DIB. This assumption made all the pieces to the puzzle come together. In demonstrating how to read a DIB you're going to read an image and color table in from a Windows .BMP file to produce the game background shown in Figure 2-9.



The Five Steps to Reading a DIB in from a .BMP File

There are five major steps to reading a DIB, as shown in Figure 2-10. A DIB file is a special format, which is a specific pattern of bytes. The reading of a DIB requires identifying certain positions or data structures in the file as you progress through it. I identified these positions as *states*. In the member function, ReadFile(), the file moves from state to state as it reads in data. (See Figure 2-11 for state diagrams.)



FIGURE 2-9

Screenshot of Example 2 displaying a DIB

1. Read the BITMAPFILEHEADER
   - verify its a DIB, does **bfType** = 'BM'?
2. Read the BITMAPINFOHEADER
3. Create the BITMAPINFO structure
   - determine the size of BITMAPINFO
   - allocate memory for BITMAPINFO
   - copy the BITMAPINFOHEADER into BITMAPINFO
4. Set up the RGBQUAD table
   - get number of colors in the table from the BITMAPINFOHEADER
   - determine the size of color table
   - read the RGBQUAD into the BITMAPINFO structure
   - assign a pointer to the RGBQUAD table in BITMAPINFO
5. Read in the image bits

FIGURE 2-10

Reading a Windows .BMP file

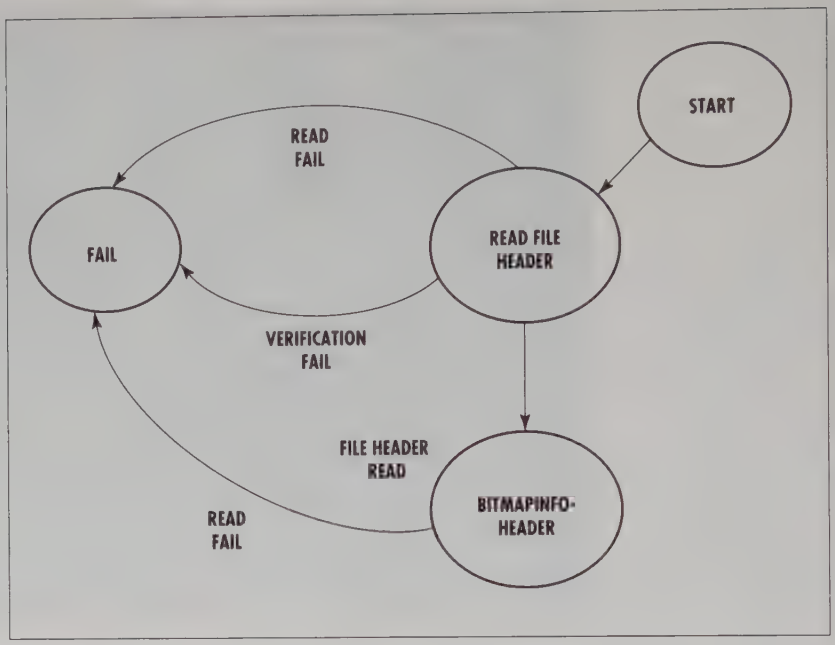


FIGURE 2-11
State tables for reading a .BMP file

To set a state in the CDib class, I added a member function SetCDibState(), which requires an enumerated state data type as input. The CDib class also has a state data member.

```
void CDib::SetCDibState( CDibState estate )
{
  state = estate;
}
```



Step 1. Reading the BITMAPFILEHEADER

The first pattern you're looking for in the DIB file is the BITMAPFILEHEADER structure, which is a fixed size. If a _lread() operation successfully returns the number of bytes read and they equal the structure's size, the program is ready to move on to the next state. If the returned bytes' read value and the BITMAPFILEHEADER size don't match, then this state has failed. The BITMAPFILEHEADER contains information on file size, DIB type, and the offset to the image bits, as shown in Figure 2-12. If the BITMAPFILEHEADER *bfType* does not contain the value 'BM', then the state will also fail:

```
BITMAPFILEHEADER   bif;
// read in the BITMAPFILEHEADER
if ((_lread(fh,&bif, sizeof(bif))) != sizeof(bif))
{
  SetCDibState(FAIL_FHDR_READ);
  _lclose(fh);
  return FALSE;
}
// is the BITMAPFILEHEADER field bfType = 'BM' ?
if (bif.bfType != 0x4D42)
{
  SetCDibState(FAIL_BFTYPE_INVALID);
  _lclose(fh);
  return FALSE;
}
```



Step 2. Reading the BITMAPINFOHEADER

The next state that member function ReadFile() moves to is the reading of the BITMAPINFOHEADER. This instance occurs after the BITMAPFILEHEADER has been retrieved and validated. The progress of this state is monitored by the return value of _lread() as well. If the value returned by the read operation doesn't match the size in bytes of the BITMAPINFOHEADER, the state fails. If the size values compare, the program moves into its next state:

```
BITMAPINFOHEADER   bih;
if ((_lread(fh, &bih, sizeof(bih))) != sizeof(bih))
{
       SetCDibState(FAIL_IHDR_READ);
       _lclose(fh);
       return FALSE;
}
```

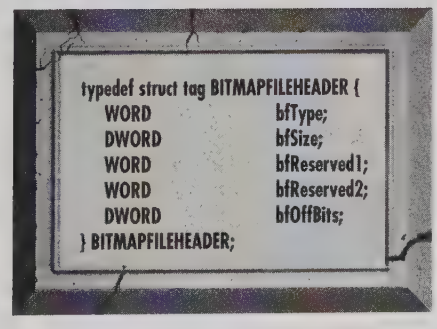


FIGURE 2-12
TYPEDEF of the BITMAPFILEHEADER



Step 3. Creating the BITMAPINFO Structure

The creation of the BITMAPINFO structure is nearly identical to what you did in the NewDIBObject() code. The state transitions dependency requires that the memory for the buffer is properly allocated. The memory required for the allocation is the size of the BITMAPHEADER plus the size of the RGBQUAD table. If the memory can't be allocated, the state fails. If the memory is allocated, the BITMAPINFOHEADER is copied into the BITMAPINFO structure. The function is ready to move to its next state.

```
biSize = sizeof(BITMAPINFOHEADER) + 256 * sizeof(RGBQUAD);
if (lpbinfo != NULL)
  GlobalFreePtr(lpbinfo);
lpbinfo = (LPBITMAPINFO) GlobalAllocPtr(GMEM_MOVEABLE,biSize);
if (lpbinfo == NULL)
{
  SetCDibState(FAIL_LPBI_ALLOC);
  _lclose(fh);
  return FALSE;
}
memset(lpbinfo, 0, biSize);
memcpy(lpbinfo, &bih, sizeof(BITMAPINFOHEADER));
```



Step 4. Reading the RGBQUAD Color Table

The BITMAPINFO structure, which now contains a copy of the BITMAPINFOHEADER, has two fields, *biBitCount* and *biClrUsed*. Both provide information about the color table in the DIB. Field *biBitCount* identifies the number of bits per pixel, and *biClrUsed* stores the value of the number entries in the RGBQUAD table. A member function, NumberOfColorEntries(), is added to the class to help provide quick retrieval of this information. This comes in useful when working with the RGBQUAD and the BITMAPINFO structure. In order to retrieve the palette from the structure, you need to know how many entries are in the table and its offset into the BITMAPINFO structure of the table's starting address. The state transition dependencies on the RGBQUAD table is the byte comparison returned from the _lread() function. If the comparisons are not equivalent the state fails. The following code achieves this:

```
nPalEntries = NumberOfColorEntries((LPBITMAPINFO) &bih);
rgbQuadTableSize = nPalEntries * sizeof(RGBQUAD);
if ((_lread(fh,((LPBYTE) lpbinfo)
➥+sizeof(BITMAPINFOHEADER),rgbQuadTableSize)) != rgbQuadTableSize)
```




```
{
        SetCDibState(FAIL_FORMAT_INVALID);
        _lclose(fh);
        return FALSE;
}
lprgb = GetLPRPGPtr();
```



Step 5. Reading the Image Bits

Allocating memory for image bits and reading them into the buffer is the final state for reading the DIB file format. The logical place to start when allocating memory is to determine how much memory is needed for the allocation. The BITMAPFILEHEADER fields *bfSize* and *bfOffset* provide you with all the information you need to know about the size of the image. The value *bfSize* is the size of the DIB file and *bfOffset* is the offset from the start of the file to the start of the image bits. The size of the image bits is simply the result of subtracting *bfOffset* from *bfSize*. With this value the memory required for the image can now be allocated. If the memory is allocated successfully, then function can proceed to read the image into memory. Otherwise, the state fails. A second state dependency checks to ensure that the image bytes read are equivalent to the size of the image buffer. If they do not match, then the state fails. The code for reading the image bits follows:

```
if (lpBits != NULL)
  GlobalFreePtr(lpBits);
lpbitsSize = bif.bfSize - bif.bfOffBits;
lpBits = (LPBYTE) GlobalAllocPtr(GMEM_MOVEABLE,lpbitsSize);
if (lpBits == NULL)
{
  SetCDibState(FAIL_BITS_ALLOC);
  _lclose(fh);
  return FALSE;
}
_llseek(fh, dwFileStart + bif.bfOffBits, FILE_BEGIN);
if ((_lread(fh, lpBits, lpbitsSize)) != lpbitsSize)
{
  SetCDibState(FAIL_BITS_READ);
  _lclose(fh);
  return FALSE;
}
```



Reading a DIB File Format in Review

Reading a DIB is not as difficult as most source code leads you to believe. Reading a DIB requires five steps: reading the BITMAPFILEHEADER, reading the BITMAPINFOHEADER, creating the BITMAPINFO structure, retrieving

the RGBQUAD color table, and retrieving the image bits. The progress of the DIB file reader is easily described using states. The overall behavior of the ReadFile() member function can be viewed as moving from state to state as it reads in the file information. Each state helps to resolve any ambiguity for reading in the bytes and allocating memory for the buffers.



Reading and Displaying a DIB Using Your Generic Game Application

Your Windows 95 Generic Game Application provides a good basis for either crafting a game or for putting together a quick and dirty application driver. I have provided the complete source code (see Listing 2-2) for reading in a DIB from a .BMP file and displaying its contents on the Windows 95 Generic Game Applications DC. The source code is useful for reference, experimenting, and other self-enlightening purposes.

LISTING 2-2 Using the Generic Game Application and CDib to display a .BMP file

```
BOOL InitGameApp( HINSTANCE hThisInstance, int nCmdShow )
{
  HWND        hwnd;
  WNDCLASS    wndclass;
  char szAppName[] = "Generic Game SDK Window";

  wndclass.style           = CS_DBLCLKS;
  wndclass.lpfnWndProc     = WndFunc;          // window function
  wndclass.cbClsExtra      = 0;                // no extra count of bytes
  wndclass.cbWndExtra      = 0;                // no extra count of bytes
  wndclass.hInstance       = hThisInstance;    // this instance
  wndclass.hIcon           = LoadIcon (NULL, IDI_APPLICATION);
  wndclass.hCursor         = LoadCursor (NULL, IDC_ARROW);
  wndclass.hbrBackground   = GetStockObject (WHITE_BRUSH);
  wndclass.lpszMenuName    = NULL;
  wndclass.lpszClassName   = szAppName;

  RegisterClass (&wndclass);
  hwnd = CreateWindowEx (
       WS_EX_TOPMOST,                    // window style
       szAppName,                        // window class name
       "Generic Windows 95 Game App",    // window caption
       WS_VISIBLE | WS_POPUP,            // window style
       CW_USEDEFAULT,                    // initial x position
       CW_USEDEFAULT,                    // initial y position
       640,                              // initial x size
       480,                              // initial y size
       NULL,                             // parent window handle
```

```
      NULL,                          // window menu handle
      hThisInstance,                 // program instance handle
      NULL);                         // creation parameters

    if (!hwnd)
      return FALSE;

  pDib = new CDib;
  if (pDib->ReadFile("scene.bmp") == FALSE)
  {
    MessageBox(hwnd,"scene.bmp file not found", "Spells of Fury", MB_OK);
    return FALSE;
  }
  ShowWindow (hwnd, nCmdShow);
  UpdateWindow (hwnd);
  SetFocus(hwnd);
  ShowCursor(FALSE);
  return TRUE;
}

//////////////////////////////////////
// WndProc for Message Processing
//////////////////////////////////////
LRESULT APIENTRY WndFunc(HWND hwnd, UINT message, WPARAM wParam,LPARAM
➥lParam)
{
  switch (message)
  {
    case WM_PAINT:
    {
      HDC         hdc;
      PAINTSTRUCT ps;

      hdc = BeginPaint (hwnd, &ps);
      HPALETTE hPalette = pDib->CreateLogicalPalette();
      HPALETTE hCurrentPalette;
      SelectPalette(hdc, hPalette, FALSE);
      RealizePalette(hdc);
      pDib->DrawBits(hdc, 0, 0);
      SelectPalette(hdc, hCurrentPalette, FALSE);
      DeleteObject(hPalette);
      EndPaint (hwnd, &ps);
    }
    return 0;
    case WM_KEYDOWN:
      switch(wParam)
      {
        case VK_ESCAPE:
          PostQuitMessage(0);
          return 0;
      }
      return 0;
```

continued on next page

continued from previous page

```
    case WM_DESTROY:
      PostQuitMessage (0);
      return 0;
  }
  return DefWindowProc (hwnd, message, wParam, lParam);
}
```

You've covered a great deal of ground since your CDib class's original design conception. You now have the means to create a DIB object from scratch as well as read a .BMP file into your application. Now let's add some serious graphics tools to your bag of tricks.



Manipulating Bits in an Offscreen Buffer

A very important factor in video animation is smooth flicker-free screen updating. Flicker occurs when a sprite is repeatedly drawn over a background or there is a timing issue when you choose to update the screen too quickly. Game programmers have resolved this problem by drawing updates to an offscreen buffer. When the drawing of the background and the sprite updates is completed, the entire offscreen buffer is written to the screen. In the not too distant past, this meant a lot of assembly language code to get the desired speed for updating a screen from an offscreen buffer. Nowadays, the programmer can be lazy and rely on the hardware to provide the speed that is necessary. This point will become clearer when you start working with DirectDraw APIs.



Understanding the Organization of DIB Bits

The primary functionality of the offscreen buffer is to accept image bits from different external buffers, such as background buffers and sprite buffers, to compose an image for a screen DC update. The process can be regarded as DIBs updating an offscreen buffer (DIB) for a device context update (see Figure 2-13). The CDib class already provides a member function for updating the device context. What you need to do is add a member function that copies the contents of one DIB object to another.

When I initially approached the problem of copying bits from DIB to DIB, I relied on one of the old Programmer's Proverbs. I don't believe they're numbered like the Ferengi Rules of Acquisition so I'll just make reference to the previous proverb quoted at the opening of this topic. Searching through old code, I already had a class I designed a few years back to copy pixels from one

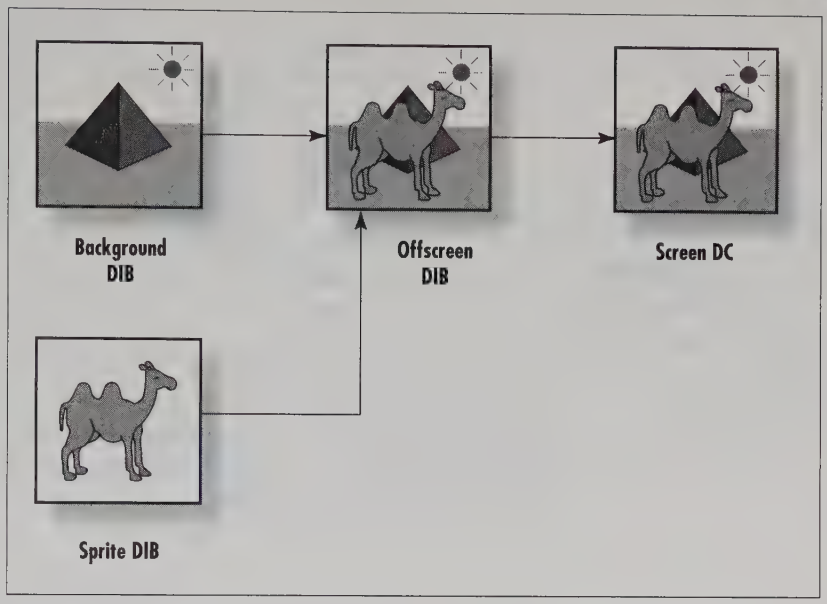


FIGURE 2-13
Composing an image with an offscreen DIB

buffer to another. Using the old code from a class I called PixBuf to handle raw bits and RGB triple palettes for PCX and IFF files, I proceeded to cut, paste, and pillage the code. Sticking to industry standards, any code that compiles with no errors is immediately pushed to alpha test. The code worked rather well, that is for copying entire DIBs of equal sizes. Where it failed was in copying images from one DIB to another using logical rectangles. Using my old code for the PCX and LBM images produced undesirable results on DIB images when I tried to copy the upper left corner, about 32 pixels by 32 pixels, to an offscreen DIB. The program copied the entire bottom 32 scanlines to the offscreen DIB. At least, this is what was thrown out to the screen DC from the offscreen DIB. Whoa, better call the project manager and tell him I'm going to miss the software release date.



Bottoms Up or...What to Do When Your Old PCX Code Fails

So what went wrong? Simple, DOS programmers are used to dealing with top-down bitmaps. PCX and LBM bitmaps have their origins (coordinates 0,0) in the logical upper left corner of the bitmap. Microsoft and IBM some time ago thought that the ways of the graphics world would turn to bottom-up bitmaps. A bottom-up bitmap has its logical origin in the lower left corner of the bitmap (see Figure 2-14).

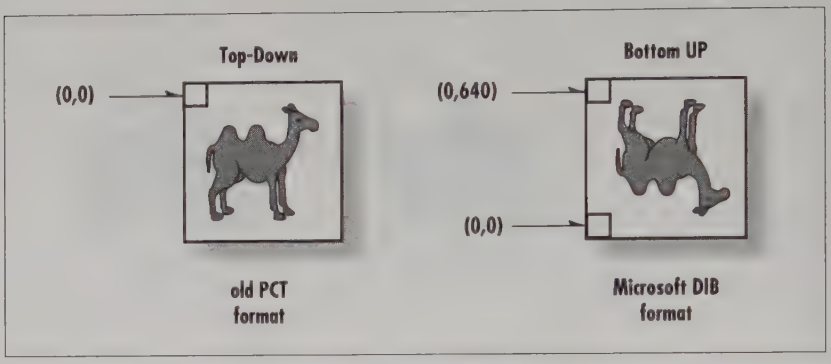


FIGURE 2-14
Top-down and bottom-up orientation of a 640 x 480 image

The solution to fix the code is simple. However, for an old DOS programmer, I was having more difficulty in tuning my mind to going in the opposite direction. Remember, the video game and the rest of the code are dealing with a bitmap that is logically organized top-down. The trick here is to provide a member function in class CDib that handles bottom-up bitmaps internally and treats them as top-down externally. This will require providing a few translations between coordinates and understanding how to navigate the image bits in the DIB. For example, to copy an entire DIB to another DIB of equal size, you could merely loop and copy bytes, like this:

```
UINT i,j;
LPBYTE lpSrcBits, lpDstBits;
for (i = 0; i < biHeight; i++)
{
  for (j = 0; j < biWidth; j++)
    lpDstBits[j] = lpSrcBits[j];
  lpDstBits += biWidth;
  lpSrcBits += biWidth;
}
```

Notice the bits' pointers are incremented by the bitmap width to reference the next row of bytes. Using this mechanism, you are actually starting from the upper left corner of the image. This is where all the mental confusion begins. The pointer referencing the image bits is actually referencing coordinate (0, biHeight) and not coordinate (0,0) like a PCX image pointer.

This code can easily be added to the class CDib in this raw form. Although it isn't particularly useful like this, it is a good starting point to build upon. The code for member function BOOL CDib::CopyDibBits() follows:

```
BOOL CDib::CopyDIBBits( const CDib& srcDIB)
{
  LPSTR srcBits, dstBits;
  srcBits = (LPSTR)srcDIB.lpBits;
  dstBits = (LPSTR)lpBits;
  unsigned int i,j;
  for(i=0; i<480;i++)
  {
```




```
    for (j=0;j<640;j++)
      dstBits[j] = srcBits[j];
    srcBits += 640;
    dstBits += 640;
  }
  return TRUE;
}
```

This source code is only useful for copying DIBs of identical sizes, namely 640 x 480 DIBs. You're going to fix this limitation soon. I don't want to entirely overload your gray cells just yet, because there is one other thing you have to do when copying DIBs. You need to grab the BITMAPINFO structure from the source DIB. This can be achieved by adding the member function Copy BITMAPINFO. This will allow a source DIB object to give a destination DIB object its BITMAPINFO structure. Why do you need to do this? Because the destination DIB object has no idea of the color palette's composition. The code for class member BOOL CDib:: CopyBITMAPINFO looks like this:

```
BOOL CDib::CopyBITMAPINFO( const CDib& srcDIB )
{
  UINT biSize;
  biSize = GetBITMAPINFOSize();
  CreateBITMAPINFO();
  memcpy(lpbinfo, srcDIB.lpbinfo, biSize);
  return TRUE;
}
```

This code provides you with the two mechanisms you need to provide a simple offscreen DIB object.



Implementing an Offscreen DIB in the Windows 95 Generic Game Application

Now after getting a bit sidetracked with definitions and logistics, you want to design an offscreen DIB object that will be a staging ground for composing backgrounds and blitting sprites before you update the DC. The idea is to update an offscreen buffer where the user never sees the updating. When the offscreen is finished being updated, it updates the DC. Implementing the offscreen DIB is really simple. You will use your illustrious Windows 95 Generic Game Application to create and display the offscreen buffer. To do this you need to add to the code two global variables representing the CDib objects. This declaration is as follows:

```
CDib* pDibBckGrnd;    // DIB holding background image
CDib* pDibOffscreen;  // offscreen DIB to draw to DC
```

You need to provide code to initialize the objects. The CDib object, *pDibBckGrnd*, needs to read in the .BMP file and this information needs to be passed along to the offscreen CDib object. The code looks like this:

```
BOOL InitDIBObjects()
{
  pDibBckGrnd = new CDib;
  pDibOffscreen = new CDib;
  if (!pDibBckGrnd->ReadFile("bckgrnd.bmp"))
    return FALSE;
  if (!pDibOffscreen->CopyBITMAPINFO( *pDibBckGrnd ))
    return FALSE;
  pDibOffscreen->CopyDIBBits( *pDibBckGrnd );
  return TRUE;
}
```

The function called InitCDibObjects() needs to be called from Windows 95 Generic Game Application function InitGameApp(). A code snippet for InitGameApp() is listed below:

```
if (!hwnd)
  return FALSE;
if (!InitDIBObjects())
  return FALSE;
ShowWindow (hwnd, nCmdShow);
UpdateWindow (hwnd);
SetFocus(hwnd);
ShowCursor(FALSE);
return TRUE;
```

If the call to InitDIBObjects() is successful, the pointer object *pDibBckGrnd* is referencing an object that contains the background image bits and BITMAPINFO structure. This information is then given to the *pDibOffscreen* pointer object.

The next item to take care of is the actual updating of the screen DC. This will require an additional function call in the Windows 95 Generic Game Application. The function UpdateScreen() will set the RGBQUAD palette from the *pDibBckGrnd* object and save the current palette in use. When this is achieved the image bits in the offscreen DIB can be updated to the screen DC. Concluding the screen update, the original palette is then restored. The code for UpdateScreen() follows:

```
void UpdateScreen(HDC hdc)
{
  HPALETTE hPalette = pDibBckGrnd->CreateLogicalPalette();
  HPALETTE hOldPalette;    // save current palette
  SelectPalette(hdc, hPalette, FALSE);
  RealizePalette(hdc);
  pDibOffscreen->DrawBits(hdc,0,0);
  SelectPalette(hdc, hOldPalette, FALSE);
  DeleteObject(hPalette);
}
```

This code is neatly packed into the WinFunc() callback function. It is called when a WM_PAINT message is processed.

```
case WM_PAINT:
{
  hdc = BeginPaint (hwnd, &ps);
  UpdateScreen(hdc);
  EndPaint (hwnd, &ps);
}
break;
```



Manipulating Bits Offscreen in Review

You discovered DIB images are stored bottom-up in memory. The image bits pointer references the upper left corner of the DIB, unlike PCX and LBM file formats whose upper left corner is at logical coordinates (0, 0). The upper left corner of the DIB has logical coordinates (0, biHeight).

The background DIB object is the master image for updating backgrounds on the offscreen DIB object. The offscreen DIB is composed of the image with which you are about to update the screen DC. It is volatile and will be drawn over, erased, or whatever is needed to compose the next screen image. The master background image is used only for copying. The sprites are messy little pixel critters who always leave traces of themselves behind. Clean up these traces by copying the portion of the master image back over the offscreen image after you have updated the DC. This will remove traces of the sprite in its old position so you can move the sprite to a new position on the screen. Figure 2-13 illustrates the idea of the offscreen DIB composing the image for the screen DC update.

To copy an image from DIB to DIB, you must copy the source DIB's BITMAPINFO structure and the image bits to the destination DIB. The destination DIB object needs to have the save RGBQUAD table to properly display the image. The BITMAPINFO structure can be copied with the CDib class member function CopyBITMAPINFO(). The image bits may be copied to a DIB object with the CDib class member function CopyDIBBits().



The BITMAPINFO structure needs to be copied to the destination object only once if the same palette is going to be used throughout the application. Shifting between palettes requires frequent updating of the destination DIB object's BITMAPINFO structure.

Screen updates are handled by the WinFunc() callback function. When WinMain() receives the message WM_PAINT, the screen is redrawn.



Example 1: Creating an Offscreen DIB Object

Example 1 demonstrates reading in a .BMP file into a master background DIB. The contents of the master background are copied to an offscreen DIB, which is used to update the screen DC. The source code for creating an offscreen CDib object is presented in Listing 2-3.

LISTING 2-3 Complete listing for creating an offscreen DIB

```
//////////////////////////////////////
// initialize DIB objects for game app
//////////////////////////////////////
BOOL InitDIBObjects()
{
  // create ptrs to DIB objects
  pDibBckGrnd = new CDib;
  pDibOffscreen = new CDib;

  // read in the DIB background
  if (!pDibBckGrnd->ReadFile("bckgrnd.bmp"))
    return FALSE;

  // copy the BITMAPINFO structure to the offscreen DIB
  if (!pDibOffscreen->CopyBITMAPINFO( *pDibBckGrnd ))
    return FALSE;
  RECT rc;
  rc.left = 0;
  rc.top = 0;
  rc.right = pDibBckGrnd->GetCDibBiWidth();
  rc.bottom = pDibBckGrnd->GetCDibBiHeight();

  // copy the background image to the offscreen DIB
  pDibOffscreen->CopyDIBBits( *pDibBckGrnd, rc, rc );
  return TRUE;
}

//////////////////////////////////////
// update the screen DC
//////////////////////////////////////
void UpdateScreen(HDC hdc)
{
   // set a palette from the background BITMAPINFO structure
   HPALETTE hPalette = pDibBckGrnd->CreateLogicalPalette();
   HPALETTE hOldPalette;    // save current palette

   // Select the background DIB palette
   SelectPalette(hdc, hPalette, FALSE);
   RealizePalette(hdc);
```

```
    // draw the image bits
    pDibOffscreen->DrawBits(hdc,0,0);

    // restore the old palette
    SelectPalette(hdc, hOldPalette, FALSE);
    DeleteObject(hPalette);
}
```

Reflecting on What You've Learned...

Your tool bag now contains software to read a .BMP file into memory and to copy two or more DIBs of equal size between each other. Now you need to work on providing functionality for sprites.

Sprite Rendering

In the previous section you used a crude means to copy one DIB to another offscreen DIB of the same size. This is fine for copying backgrounds from one image buffer to the offscreen buffer, but it is highly inadequate for drawing sprites. You need to refine the CopyDIBBits() member function to copy DIB objects of arbitrary shape, namely sprites. A sprite is an image typically much smaller than the background; for example, a Boss villain in an arcade game would be a sprite or composed of several sprites (see Figure 2-15).

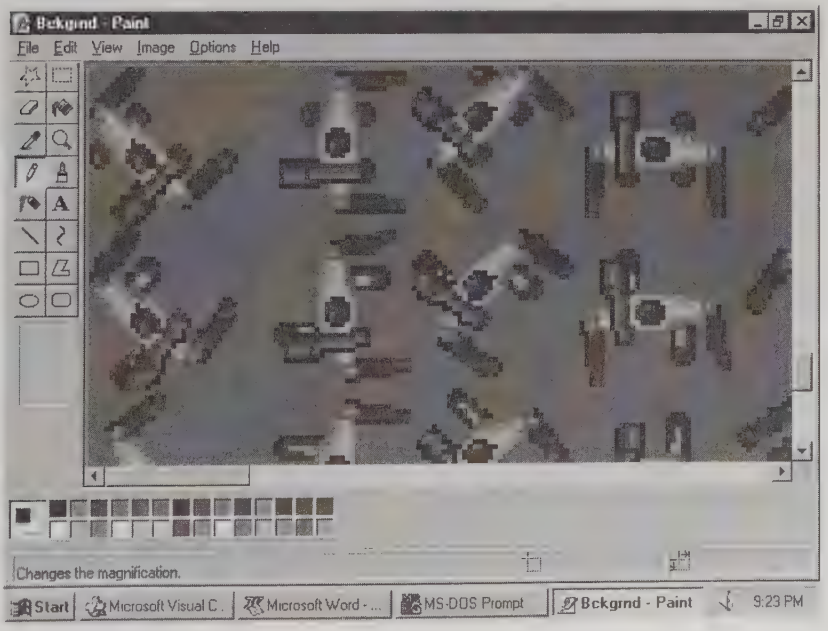


FIGURE 2-15

Sprites are small screen objects of arbitrary shape

It is common in video games for a sprite to have dimensions of 32 pixels x 32 pixels. Ken Nicholson at MediaVision informed me that this is the dimension of the sprites used in their port of Accolade's Bubsy, using the Windows 95 Game SDK. Sprites that are 32 x 32 pixels are divisible by four; hence they can be word aligned for optimum performance.



Retrieving a Sprite from a DIB

You have a minor software engineering dilemma on your hands. You can copy a DIB 640 x 480 to a DIB of size 640 x 480. Likewise, you can copy a DIB of 32 x 32 to a DIB of size 32 x 32. What your software doesn't provide is a means to copy a 32 x 32 sprite image from a master sprite DIB, say, 320 x 240 pixels in size, to an offscreen DIB 640 x 480 pixels in size. Should you notify your manager of another setback? If you do, you better bring lots and lots of cheese to the marketing department's whine fest. Or...you can be a hero and fix the code you've already got. I prefer the latter. The rewards are typically reflected in the paycheck rather than a ten-second eject out onto the parking lot asphalt.

Currently you have the following code for member function CopyDIBBits():

```
BOOL CDib::CopyDIBBits( const CDib& srcDIB)
{
  LPSTR srcBits, dstBits;
  srcBits = (LPSTR)srcDIB.lpBits;
  dstBits = (LPSTR)lpBits;
  unsigned int i,j;
  for(i=0; i<480;i++)
  {
    for (j=0;j<640;j++)
      dstBits[j] = srcBits[j];
    srcBits += 640;
    dstBits += 640;
  }
  return TRUE;
}
```

The first mechanism you need to provide is a means to logically describe the source image location and the destination image location in their respective DIBs. You will use the Windows API data structure RECT to make this distinction. For this example, you will not assume any resizing or scalability in the DIB bits copying. That is, if the source image is 32 x 32 pixels, then the destination will be 32 x 32 pixels as well. The member function CopyDIBBits() now has the following parameters for its calling arguments:

```
BOOL CopyDIBBits( const CDib& srcDIB, RECT srcRect, RECT dstRect)
```

To add more functionality, I also added the data member RECT bounds to the CDib class. Since copying bits is going to be the primary operation of DIB

objects, it makes sense to have a data member easily accessible by using the C++ *this* mechanism.



For those of you just getting your feet wet in the C++ programming pool, I direct your attention to Stanley Lippman's *C++ Primer*. Member functions access their data members through an implicit pointer to the object. The pointer's name is the *this* pointer. The *this* pointer is typically transparent in data member code and is implied. The *this* pointer is accessible to the programmer. A typical usage for the *this* pointer is when you're returning the address of an object to an object pointer.

Using the RECT data structure, the sprite dimensions can be calculated from the srcRect parameter.

```
int srcHeight = srcRect.bottom - srcRect.top;
int srcWidth = srcRect.right - srcRect.left;
```

These calculations provide some information on the size and width of the sprite. Why do you need this? Recall that the srcDIB object could be 640 x 480 pixels in size and you only care about a sprite image 32 x 32 pixels inside this DIB. These computations provide information to treat the sprite as a logical buffer.

For copying images from a buffer of one size to a buffer of a non-similar size, you need to retain information about the actual physical sizes of the source and destination buffers. The source and destination scanline width can be computed from the DIB object's data member RECT bounds (see Figure 2-16):

```
srcScanLines = srcDIB.bounds.right - srcDIB.bounds.left;
dstScanLines = bounds.right - bounds.left;
```

You now have enough information to rewrite your original CopyDIBBits() function:

```
srcBits = (LPSTR)srcDIB.lpBits;
dstBits = (LPSTR)lpBits;
UINT srcIndex, dstIndex;    // indexes into the DIB's image bits
int row,col=0;
for(row=0; row<srcHeight; row++)
{
  for (col=0;col<srcWidth;col++)
  {
    srcIndex = col + srcScanLines;
    dstIndex = col + dstScanLines;
    dstBits[dstIndex] = srcBits[srcIndex];
  }
}
```

Again, this is precisely the original function CopyDIBBits(), reorganized to handle copying images from DIBs of equal sizes. The problem of only being able

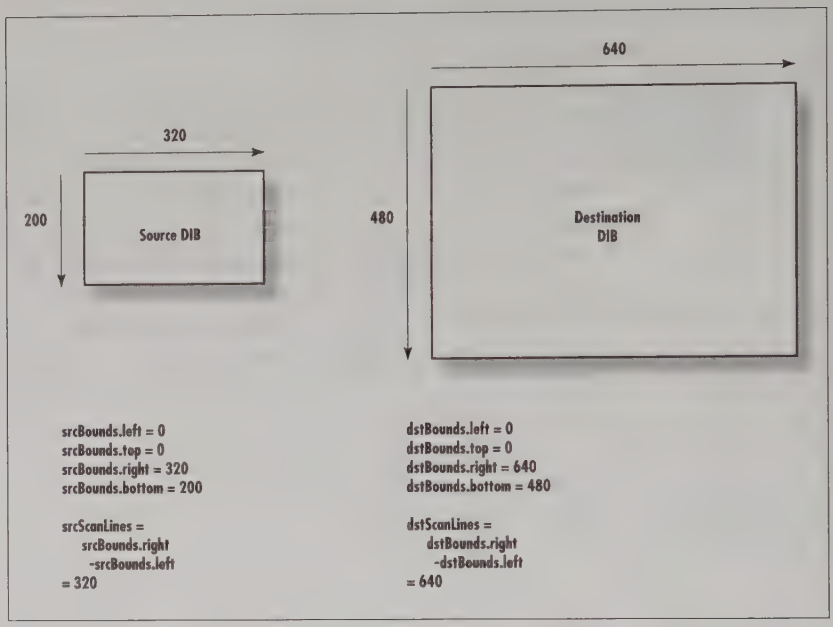


FIGURE 2-16
Usage of logical bounding rectangles

to copy DIBs of the same size still exists. No need to page the project manager yet. You didn't code yourself into a hole. This code is a little more flexible than the original version. This format allows you to play games with the bits pointer indexes. The problem of copying image bits from the upper left corner is still with you too. The image bits pointers are both referencing pixels at coordinates (0, biHeight), as shown in Figure 2-14. You need to resolve this so that the loop examines the image from the DIB's logical origin (0, 0). I approached the problem by writing down some coordinates in my notebook to look for a correlation. Assuming I had a 640 x 480 DIB, I wanted to find a translation between where the image pointer was located, at (0, 479), and the logical mapping to the pixel I wanted to retrieve. Simulating a loop I came up with the following relationships:

```
row = 0          srcBits->(0,479)     I want pixel (0,0) => (0, offset =
                                                            479 - 0)
row = 1          srcBits->(0,478)     I want pixel (0,1) => (0, offset =
                                                            478 - 1)
row = 2          srcBits->(0,477)     I want pixel (0,2) => (0, offset =
                                                            477 - 2)
row = n          srcBits->(0, biHeight - n - 1)
```

The relationship I needed to logically map the bottom-up DIB for retrieving the proper row was

```
row translation = scanline * (srcBitsRow - row-1)
```

where the variable scanline is the width of the row and srcBitsRow is the actual row in the bottom-up DIB scheme. As you can see, the row translation for converting a top-down coordinate to a bottom-up coordinate is actually an index into the image bits itself. To locate the desired row in the DIB, recall you have to create an index, which is the scanline, or row width, times the row translation, which is (srcBitsRow - row-1). Plugging these back into the loop I can now physically copy the contents at row 32 directly over to the row of a destination DIB keeping the proper orientation of the DIB. The loop now looks like this:

```
UINT srcDIBBiHeight = srcDIB. GetCDibBiHeight();
UINT dstDIBBiHeight = GetCDibBiHeight();
int row,col=0;
for(row=0; row<srcHeight; row++)
{
  for (col=0;col<srcWidth;col++)
  {
    srcIndex = col + srcScanLines * (srcDIBBiHeight - row - 1);
    dstIndex = col + dstScanLines * (dstDIBBiHeight - row - 1);
    dstBits[dstIndex] = srcBits[srcIndex];
  }
}
```

You now have the capability of grabbing an image of srcHeight from one DIB and copying it to a destination DIB. This is fine if you want to copy 48 rows from one DIB to 48 rows on another DIB, as shown in Figure 2-17. This isn't adequate for copying sprites that have a width and an offset into the source DIB. This is actually easy to resolve. You already located the row in the source DIB where the sprite is located; you just need to inform the column copy loop of where to draw. The same variables that were used to compute the *srcWidth* come into play again. To tell the copy loop to start drawing *n* pixels in from the left, you simply add *srcBound*, left to the *srcBits* index.

```
int row,col=0;
for(row=0; row<srcHeight; row++)
{
  for (col=0;col<srcWidth;col++)
  {
    srcIndex = col + srcScanLines
        * (srcDIBBiHeight - row - 1 - srcRect.top) + srcRect.left;
    dstIndex = col + dstScanLines
        * (dstDIBBiHeight - row - 1- dstRect.top) + dstRect.left;
    dstBits[dstIndex] = srcBits[srcIndex];
  }
}
```

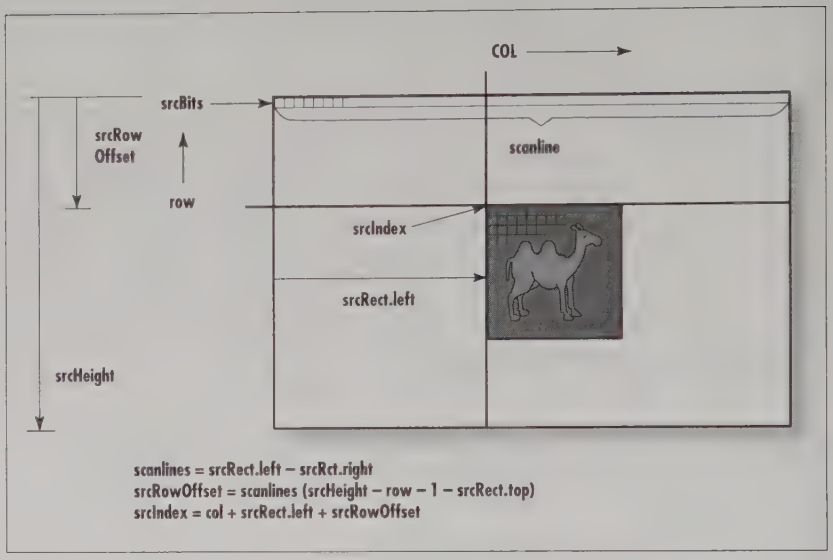


FIGURE 2-17
Locating a sprite in a DIB

The code for copying sprites is now complete. The member function CopyDIBBits() looks like this:

```
BOOL CDib::CopyDIBBits( const CDib& srcDIB, RECT srcRect, RECT dstRect)
{
  LPSTR srcBits, dstBits;
  int srcHeight, srcWidth;
  int srcScanLines, dstScanLines;
  srcHeight = srcRect.bottom - srcRect.top;
  srcWidth = srcRect.right - srcRect.left;
  srcScanLines = srcDIB.bounds.right - srcDIB.bounds.left;
  dstScanLines = bounds.right - bounds.left;
  srcBits = (LPSTR)srcDIB.lpBits;
  dstBits = (LPSTR)lpBits;
  BITMAPINFOHEADER* lpbi = (BITMAPINFOHEADER*) srcDIB.lpbinfo;
  UINT srcDIBBiHeight = (UINT)lpbi->biHeight;
  UINT srcIndex, dstIndex;
  UINT row,col=0;
  for(row=0; row<srcHeight; row++)
  {
    for (col=0;col<srcWidth;col++)
    {
      srcIndex = col + srcScanLines
            * (srcDIBBiHeight - row - 1 - srcRect.top) + srcRect.left;
      dstIndex = col + dstScanLines
            * (GetCDibBiHeight() - row - 1 - dstRect.top) + dstRect.left;
      dstBits[dstIndex] = srcBits[srcIndex];
    }
```

```
    }
    return TRUE;
}
```

To call the CopyDIBBits(), you could do the following:

```
CDib* Sprite;
CDib* Offscreen;
        :
RECT SpriteSrcRect;
RECT SpriteDstRect;

        Offscreen->CopyDIBBits( &Sprite, SpriteSrcRect, SpriteDstRect);
```

You'll notice that the DIB objects *Sprite* and *Offscreen* are already instantiated. The two RECT data structures are the source RECT of the image in the *Sprite* DIB object and its destination in the *Offscreen* object. The effect is a sprite with its bounding source rectangle, shown in Figure 2-18, which will draw over the background DIB bits at location *SpriteDstRect*.



Sprites of Arbitrary Shape and the Transparent Pixel

From a programming point of view, it's important to understand how to deal with bounded rectangular sprites. However, sprites, when they're displayed on the game screen, don't display their rectangular backgrounds as did the previous example. This problem is resolved with a trick graphics programmers



FIGURE 2-18
Example sprite with its bounding RECT visible

use in which one color of the sprite is defined as a transparency color. When the artist creates and finishes the sprite artwork, he or she selects a color not used in the image and fills the surroundings of the sprite with this color. The unused color is the transparency color. This color is essentially a flag, if you will, to tell the computer when to copy image bits. If the transparency color is encountered, this particular byte will not be drawn over the background image (see Figure 2-19). Programmers love to select black or white as their transparency values because they are easily identifiable as 0 or 255 in a color table. However, artists need black and white for gradations of light in their artwork. Artwork and cool screenshots are what persuade a consumer to purchase the software in the first place. So be kind to your artists and let them select the transparent color.

So how do you, the programmer, determine which pixel is used for transparency? Simple, let your artist know that whenever you are going to draw their *cool* sprite to the screen, you are going to check the upper left corner of the sprite to retrieve the transparency color. What you need from your DIB object is a means to examine an image byte value at a logical Cartesian coordinate in the image bits. By using the data member RECT bounds you can calculate the image scanline's offset value

```
UINT  srcScanLines = bounds.right - bounds.left;
```

The image is stored bottom-up, which means you need to do a few more calculations to index the desired pixels. The desired row index can be determined from the following:

```
UINT srcIndex;
srcIndex = srcScanLines * (GetCDibBiHeight() - y - 1);
```

The *srcIndex* value is nearly identical to the one that was calculated for CopyDIBBits(). In CopyDIBBits() you added the column as an offset into the row index to reference the desired pixel. Guess what? The x coordinate is the offset into the row index, so now *srcIndex* is calculated as follows:

```
UINT srcIndex = x + srcScanLines * (GetCDibBiHeight() - y - 1);
```

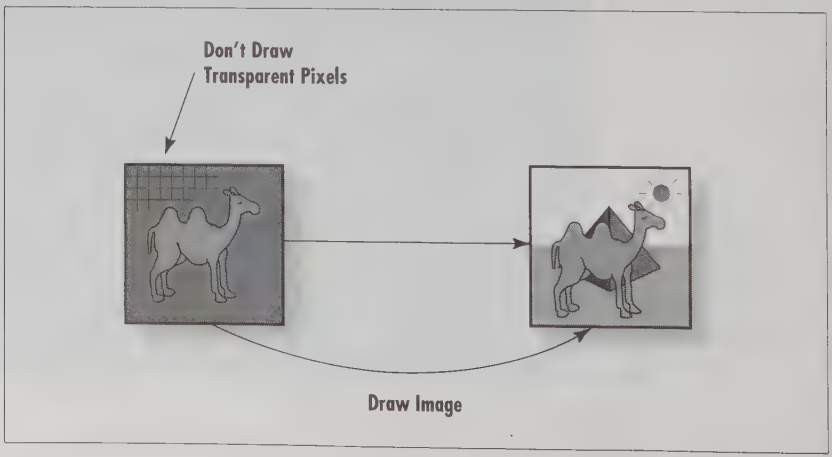


FIGURE 2-19
Using transparency values for blitting sprites



Pretty simple, huh? You can now return the byte value at the desired (x,y) location in the image bits. Wrapping this functionality back into your CDib class, you have the member function GetPixelByte(). Its listing follows:

```
BYTE CDib::GetPixelByte( UINT x, UINT y)
{
  LPSTR srcBits = (LPSTR)lpBits;
  UINT  srcScanLines = bounds.right - bounds.left;
  UINT srcIndex;
  srcIndex = x + srcScanLines * (GetCDibBiHeight() - y - 1);
  return (BYTE) lpBits[srcIndex];
}
```

The member function GetPixelByte() provides the mechanism you need to retrieve the transparency byte value from the image as well as keep your artists happy. All you have left to do is to fill in the blanks in CopyDIBBits() to allow it to handle transparency colors. Recall the copy loops from CopyDIBBits():

```
UINT srcIndex, dstIndex;
UINT row,col=0;
for(row=0; row<srcHeight; row++)
{
  for (col=0;col<srcWidth;col++)
  {
    srcIndex = col + srcScanLines
             * (srcDIBBiHeight - row - 1 - srcRect.top) +    srcRect.left;
    dstIndex = col + dstScanLines
             * (GetCDibBiHeight() - row - 1 - dstRect.top) + dstRect.left;
    dstBits[dstIndex] = srcBits[srcIndex];
  }
}
```

Now you need to add an *if* statement to check for the transparency value, which looks like this:

```
BYTE transparencyColor;
UINT srcIndex, dstIndex;
UINT row,col=0;
for(row=0; row<srcHeight; row++)
{
  for (col=0;col<srcWidth;col++)
  {
    srcIndex = col + srcScanLines
             * (srcDIBBiHeight - row - 1 - srcRect.top) +   srcRect.left;
    dstIndex = col + dstScanLines
             * (GetCDibBiHeight() - row - 1 - dstRect.top) + dstRect.left;
    if (dstBits[dstIndex] != transParencyColor)
      dstBits[dstIndex] = srcBits[srcIndex];
  }
}
```

Since CopyDIBBits() merely copies source image to destination image, I chose to add a new member function CopyTransDIBBits():

```
BOOL CDib::CopyTransDIBBits( const CDib& srcDIB, RECT srcRect, RECT dstRect,
➥BYTE transByte)
{
  LPSTR srcBits, dstBits;
  int srcHeight, srcWidth;
  int srcScanLines, dstScanLines;

  srcHeight = srcRect.bottom - srcRect.top;
  srcWidth = srcRect.right - srcRect.left;
  srcScanLines = srcDIB.bounds.right - srcDIB.bounds.left;
  dstScanLines = bounds.right - bounds.left;
  srcBits = (LPSTR)srcDIB.lpBits;
  dstBits = (LPSTR)lpBits;
  BITMAPINFOHEADER* lpbi = (BITMAPINFOHEADER*) srcDIB.lpbinfo;
  UINT srcDIBBiHeight = (UINT)lpbi->biHeight;
  UINT transColor = 0;
  UINT srcIndex, dstIndex;
  BYTE transParentColor = LOBYTE(LOWORD(transByte));
  UINT row,col=0;
  for(row=0; row<srcHeight; row++)
  {
    for (col=0;col<srcWidth;col++)
    {
      srcIndex = col + srcScanLines
          * (srcDIBBiHeight - row - 1 - srcRect.top) + srcRect.left;
      dstIndex = col + dstScanLines
          * (GetCDibBiHeight() - row - 1 - dstRect.top) + dstRect.left;
      if ( (unsigned char) srcBits[srcIndex]!= (unsigned char)transByte)
      {
        dstBits[dstIndex] = srcBits[srcIndex];
      }
    }
  }
  return TRUE;
}
```

The member functions GetPixelByte() and CopyTransDIBBits() can be used together to retrieve a transparency value and perform a transparent blit operation, as shown in Figure 2-20. This operation would look like this:

```
CDib* Sprite;
CDib* Offscreen;
:
RECT SpriteSrcRect;
RECT SpriteDstRect;
UINT x,y;
BYTE  transPixel = GetPixelByte(x,y);
Offscreen->CopyTransDIBBits( &Sprite, SpriteSrcRect, SpriteDstRect,
➥transPixel);
```

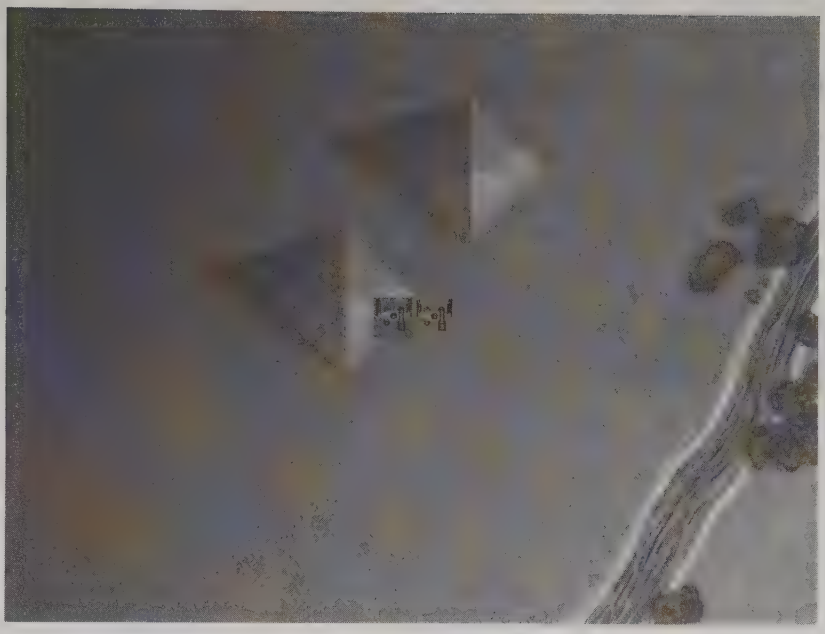


FIGURE 2-20
Displaying a transparent sprite and a nontransparent sprite



Retrieving a Sprite from a DIB Review

A sprite is a small screen object of arbitrary shape. Sprites are stored in offscreen buffers whose source and destination locations are described by rectangular coordinates. An optimum size for a sprite in a DIB is 32 x 32 pixels. This is because DIBs are 32-bit DWORD-aligned.

Sprites can be copied to the screen directly using the CDib member function CopyDIBBits(). This call is useful for tiling, where the bounding rectangle is a component of the shape itself.

Copying a sprite of arbitrary shape to the screen, such as a spaceship, or a warrior, requires the transparent pixel testing mechanism. This involves testing to see if the pixel value is equal to the transparent pixel value. If the values are equal, this pixel is skipped and not drawn onto the background. The CDib member function GetPixelByte() is used to retrieve and set the transparent pixel value from the DIB. This allows arbitrary values to be used as the transparent pixel value. The CDib member function, CopyTransDIBBits(), copies a sprite using the transparent pixel testing scheme.

Example 2: Drawing Sprites to the DC

Well you've made it. Your *Fisher Price* graphics interface is complete. This demonstration draws two sprites, one using transparency, and the other just drawing over the background. The purpose is to illustrate the usage of transparency for drawing sprites on backgrounds. The complete source code is provided in Listing 2-4.

LISTING 2-4 InitDIBObjects in Example 4 performs transparency blitting

```
BOOL InitDIBObjects()
{
  pDibBckGrnd = new CDib;
  pDibOffscreen = new CDib;

  // offscreen DIB rect
  RECT offscreenRect;
  offscreenRect.left = 0;
  offscreenRect.top = 0;
  offscreenRect.right = 640;
  offscreenRect.bottom = 480;
  pDibBckGrnd->ReadFile("bckgrnd.bmp");
  pDibOffscreen->CopyBITMAPINFO( *pDibBckGrnd );

  // 640 x 720 background DIB is larger than offscreen DIB
  // restore the 640 x 480 dimensions
  pDibOffscreen->SetBitmapHeader(640,480);

  // copy the background image to the offscreen DIB
  pDibOffscreen->CopyDIBBits( *pDibBckGrnd, offscreenRect, offscreenRect );

  // read in the sprites
  RECT spriteRect;
  spriteRect.left = 0;
  spriteRect.top = 480;
  spriteRect.right = 32;
  spriteRect.bottom = 512;

  // destination rect for non-transparent sprite copy
  RECT destRect;
  destRect.left = 288;
  destRect.top = 224;
  destRect.right = 320;
  destRect.bottom = 256;

  // non-transparent copy sprite from bckgrnd DIB to offscreen
  pDibOffscreen->CopyDIBBits( *pDibBckGrnd, spriteRect, destRect );
```

```
  // destination rect for transparent sprite copy
  RECT tDestRect;
  tDestRect.left = 320;
  tDestRect.top = 224;
  tDestRect.right = 352;
  tDestRect.bottom = 256;

  // use upper left corner pixel of sprite as transparent byte
  BYTE transParentPixel = pDibBckGrnd->GetPixelByte(0,500);

  // transparent copy sprite from bckgrnd DIB to offscreen
  pDibOffscreen->CopyTransDIBBits( *pDibBckGrnd,
  spriteRect, tDestRect, transParentPixel );
  return TRUE;
}
```



A CDib Object Animation Example

MJ-12 CLASSIFIED DOCUMENT
Author: 13

Preface

The Majority 12 must consider alternative countermeasures in its efforts to stop the artifact and its mission to dominate mankind. It is a matter of time before the artifact resolves network number sequencing encryptions and finds its way onto the Internet. It is uncertain whether the threat of a fissionable device is at the disposal of the executing implanted neural image of the one called Agamemnon. The magnitude of the threat must be resolved before attempting to remove network connections from the device.

Attention must also be directed at the growing number of expendables that failed to infiltrate and recon the artifact with the virtual interfaces provided. A possible solution is to interface to the device with a computer of unknown configuration in order to infiltrate Agamemnon's databases. This new computer will provide the ports by which non-expendables will commence an assault on strategic data descriptors maintaining the artifact's databases. The software for this cleansing exercise is code named AssaultWare.



PowerSuit

Soldiers entering the artifact will pilot virtual automatons through the input of special body response suits. Input of the suits will be maintained by a computing device of undisclosed origin. Unfamiliar hardware should slow

down the artifact's countermeasures and allow a window of invasive entry. One to 12 PowerSuits can be piloted simultaneously to one to 12 confidential software and hardware strategic locations.



Feasibility Study

A feasibility study was carried out on an IBM PC Pentium system running Windows 95. The emphasis of the study was to determine human response factors in piloting a PowerSuit with non-instinctive controls such as keyboard and mouse. The test was conducted on a 2D logical grid and physically limited to the resolution of the application window, measuring 640 x 480 pixels. Figure 2-21 depicts a simulated battleground of a PowerSuit prototype engaging a static target. The test assumed a location Agamemnon may select based on his familiarity with ancient Egypt.

The AssualtWare used to operate the prototype is governed by movement on a 2D logical grid. For the simplicity of this test, it is a standard Cartesian grid. The grid represents the 640 x 480 pixels in a Windows 95 Device-Independent Bitmap divided up into square units of 32 pixels. In this configuration the logical grid measures 20 units (640/32 pixels per unit) x 15 units (480/32 pixels per unit). Figure 2-22 shows a representation of the test grid. A PowerSuit moves within the grid based on keyboard commands from the numeric keypad. The keypad is mapped so keystrokes represent a direction vector within the grid. The key mappings are shown in Figure 2-23.



FIGURE 2-21

Screenshot of a PowerSuit using a standard Windows 95 Win32 interface

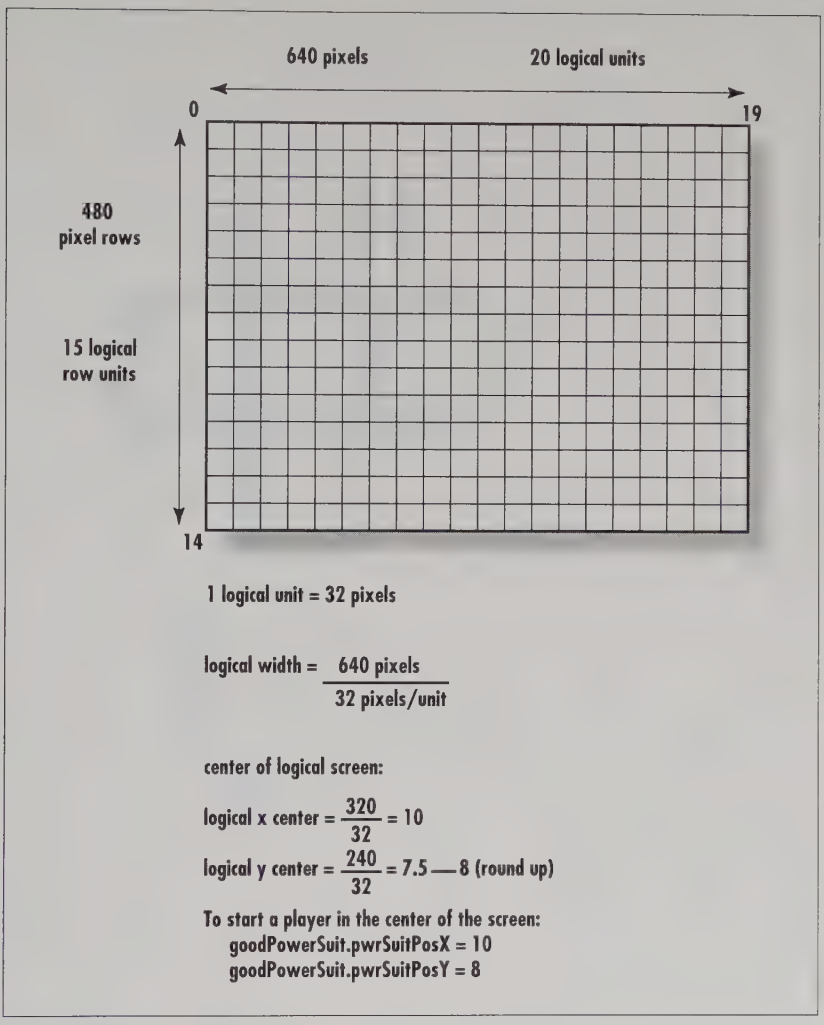


FIGURE 2-22
PowerSuit movement grid

A total of eight vectors exist to give the PowerSuit multiple degrees of freedom. The [2] and [8] keys on the keypad control forward and reverse momentum. These account for two of the direction vectors. The other six heading vectors occur during a rotation of the PowerSuit. Rotations occur by pressing the [4] key, which rotates to the left, or the [6] key, which rotates to the right. The composite vectors are shown in Figure 2-24. The vectors represented are defined in the PWRSUIT.H header file as follows:

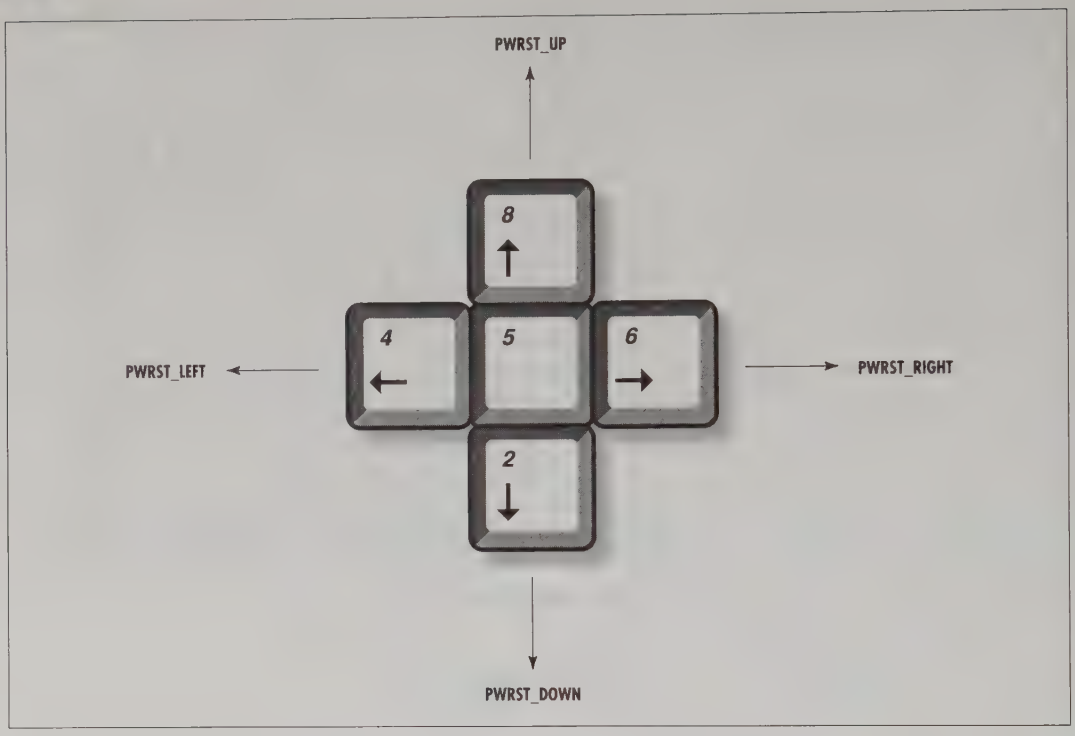


FIGURE 2-23 PowerSuit control key mappings

```
#define PWRST_UP            0
#define PWRST_UPRIGHT       1
#define PWRST_RIGHT         2
#define PWRST_DNRIGHT       3
#define PWRST_DOWN          4
#define PWRST_DNLEFT        5
#define PWRST_LEFT          6
#define PWRST_UPLEFT        7
```

The following excerpt of code demonstrates the rotation cycles of the heading values:

```
// Turn the powersuit to the left
case VK_NUMPAD4:
  goodPowerSuit.heading--;
  if (goodPowerSuit.heading < PWRST_UP)
    goodPowerSuit.heading = PWRST_UPLEFT;
  break;
// Turn the powersuit to the right
case VK_NUMPAD6:
  goodPowerSuit.heading++;
  if (goodPowerSuit.heading > PWRST_UPLEFT)
    goodPowerSuit.heading = PWRST_UP;
  break;
```

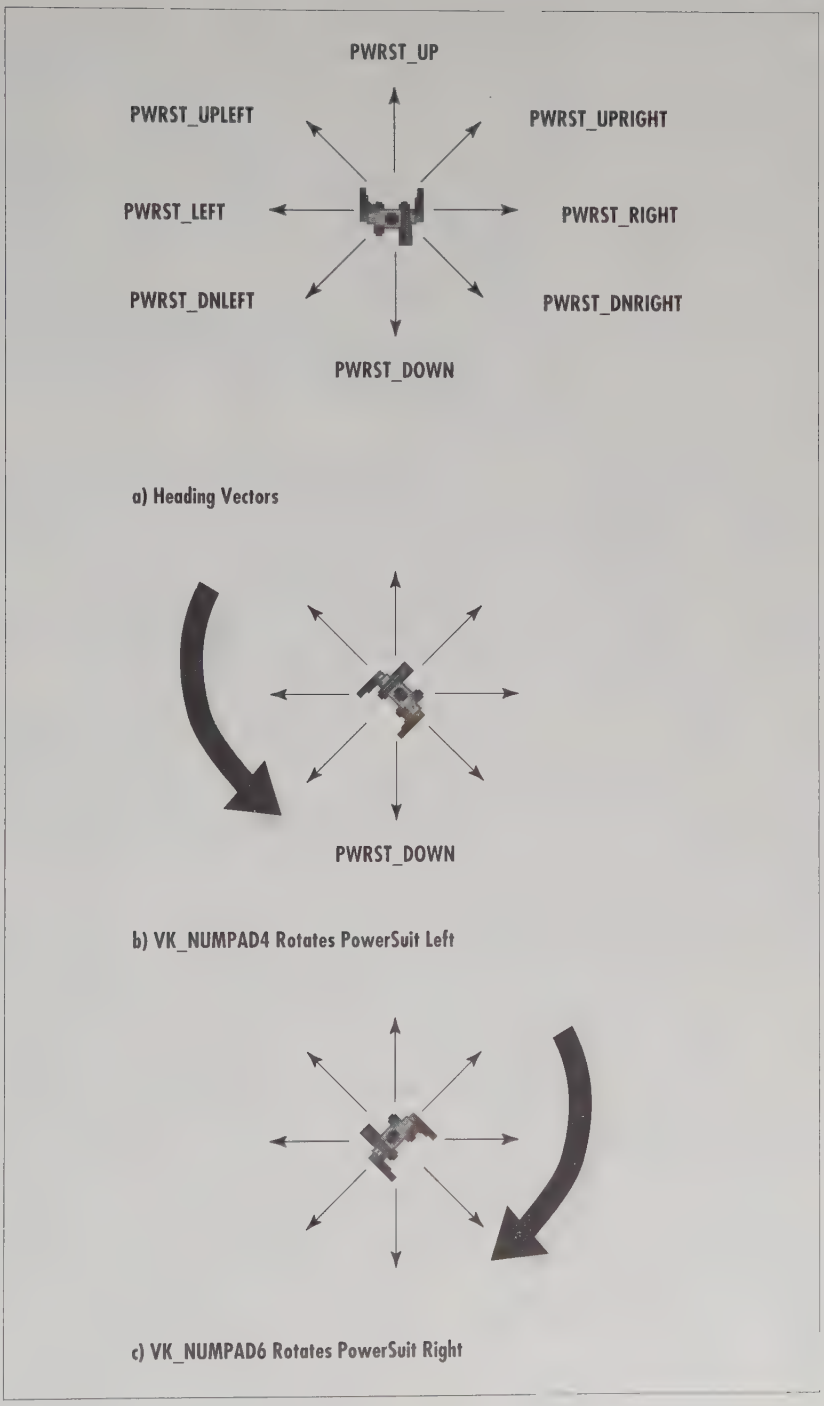


FIGURE 2-24

The eight heading vectors used in PowerSuit movement

A Windows 95 WM_KEYDOWN message with its *wParam* set to VK_NUMPAD4 causes the headings to decrement from the current heading to the next heading value. For instance, headings are values successively defined as PWRST_UP up to PWRST_UPLEFT. If the heading is currently PWRST_DNRIGHT, it will be decremented to the value of PWRSUIT_RIGHT. The values cycle counterclockwise as shown in Figure 2.24(b). If the current heading is PWRST_UP, it will wrap around to PWRST_UPLEFT when the heading is decremented.

If the *wParam* contains the value VK_NUMPAD6, the heading values are incremented and cycle in a clockwise direction. This is shown in Figure 2-24(c).

The headings and the movement values of the PowerSuit are maintained in a struct, which looks like this:

```
typedef struct PwrSuitInfoStruct
{
  UINT pwrSuitPosX;        // pwrSuitPosX and pwrSuitPosY are the
  UINT pwrSuitPosY;        // upper left corner sprite rect
  UINT pwrSuitPatchX;
  UINT pwrSuitPatchY;
  int heading;             // current sprite being drawn
  RECT spriteRect;         // Location of base for power suit's sprites
  RECT destRect;           // Location of power suit sprite on background
  RECT patchRect;          // Where to patch the background
  DWORD spriteFrame;       // current cursor frame
  UINT moveFrame;          // How far mech has moved in current move.
  int movex;               // Move mech # of pixels in x direction
  int movey;               // Move mech # of pixels in y direction
  char firingWeapon;       // Flag for when mech is firing weapon
  char spriteChange;       // Flag for when sprite has changed on-moving mech
  char hits;               // Number of hits the powersuit can still take.
} pwrSuitInfo;

pwrSuitInfo goodPowerSuit;
```

The PowerSuit undergoing the testing is represented in the structure *goodPowerSuit*. The fields *pwrSuitPosX* and *pwrSuitPosY* are the x and y coordinates for the logical grid. The PowerSuit's direction of motion is contained in the field heading. The fields *movex* and *movey* are the units of movement on the actual DIB. The PowerSuit will take four steps in each logical cell to the next. Therefore, the *movex* and *movey* values are typically ±8 pixels. Four steps times eight pixels is 32 pixel units of movement. The code to control the PowerSuit movement is handled by the function MoveSuit(). MoveSuit() accepts a pointer to a pwrSuitInfo struct and an integer for movement as its calling arguments:

```
void MoveSuit(pwrSuitInfo *pwrSuit, int movement)
{
  switch(pwrSuit->heading)
  {
```

```
    case PWRST_UP:
       pwrSuit->pwrSuitPosY-=movement;
       pwrSuit->movex = 0;
       pwrSuit->movey = -8;
       break;
    case PWRST_UPRIGHT:
       pwrSuit->pwrSuitPosY-=movement;
       pwrSuit->pwrSuitPosX+=movement;
       pwrSuit->movex = 8;
       pwrSuit->movey = -8;
       break;
    case PWRST_RIGHT:
       pwrSuit->pwrSuitPosX+=movement;
       pwrSuit->movex = 8;
       pwrSuit->movey = 0;
       break;
    case PWRST_DNRIGHT:
       pwrSuit->pwrSuitPosY+=movement;
       pwrSuit->pwrSuitPosX+=movement;
       pwrSuit->movex = 8;
       pwrSuit->movey = 8;
       break;
    case PWRST_DOWN:
       pwrSuit->pwrSuitPosY+=movement;
       pwrSuit->movex = 0;
       pwrSuit->movey = 8;
       break;
    case PWRST_DNLEFT:
       pwrSuit->pwrSuitPosY+=movement;
       pwrSuit->pwrSuitPosX-=movement;
       pwrSuit->movex = -8;
       pwrSuit->movey = 8;
       break;
    case PWRST_LEFT:
       pwrSuit->pwrSuitPosX-=movement;
       pwrSuit->movex = -8;
       pwrSuit->movey = 0;
       break;
    case PWRST_UPLEFT:
       pwrSuit->pwrSuitPosY-=movement;
       pwrSuit->pwrSuitPosX-=movement;
       pwrSuit->movex = -8;
       pwrSuit->movey = -8;
       break;
};
// Assume the powersuit is not moving.
pwrSuit->moveFrame = 5;
// If powersuit is not moving off the screen then let suit
// start moving.
if (pwrSuit->pwrSuitPosX >= MIN_GRID_X
```

```
        && pwrSuit->pwrSuitPosX <= MAX_GRID_X
        && pwrSuit->pwrSuitPosY >= MIN_GRID_Y
        && pwrSuit->pwrSuitPosY <= MAX_GRID_Y)
    pwrSuit->moveFrame = 1;
    // Make sure the numbers stay within the screen bounds.
    if(pwrSuit->pwrSuitPosX < MIN_GRID_X)
        pwrSuit->pwrSuitPosX = MIN_GRID_X;
    if(pwrSuit->pwrSuitPosY < MIN_GRID_Y)
        pwrSuit->pwrSuitPosY = MIN_GRID_Y;
    if(pwrSuit->pwrSuitPosX > MAX_GRID_X)
        pwrSuit->pwrSuitPosX = MAX_GRID_X;
    if(pwrSuit->pwrSuitPosY >MAX_GRID_Y)
        pwrSuit->pwrSuitPosY = MAX_GRID_Y;
    // This sets the direction of where the mech walks on the
    // screen
    pwrSuit->movex *= movement;
    pwrSuit->movey *= movement;
}
```

The routine MoveSuit() is called when either the 2 or the 8 key on the numeric keypad is pressed. The VK_NUMPAD2 key will issue a command to move the PowerSuit backwards. A unit movement vector of -1 represents a backwards or negative movement with respect to the facing direction of the PowerSuit. Likewise, the VK_NUMPAB8 key issues a command to move the PowerSuit forward. The unit movement vector is +1 for forward or positive motion.

To illustrate the PowerSuit movement within the logical grid, shown in Figure 2-25, you have a PowerSuit with the following conditions when the VK_NUMPAD8 key is pressed:

```
goodPowerSuit.pwrSuitPosX = 9;
goodPowerSuit.pwrSuitPosX = 9;
goodPowerSuit.heading = PWRST_DNLEFT;
```

When the routine MoveSuit is called, it will examine the PowerSuitInfo field heading to calculate the next logical grid coordinate. In the example, the field heading contains the value PWRST_DNLEFT. This indicates you wish to move the PowerSuit forward and head to the lower left corner of the logical grid. By examining the heading, you use the following code based on the case statement:

```
case PWRST_DNLEFT:
    pwrSuit->pwrSuitPosY+=movement;
    pwrSuit->pwrSuitPosX-=movement;
    pwrSuit->movex = -8;
    pwrSuit->movey = 8;
    break;
```

The PowerSuit's positions, *pwrSuitPosX* and *pwrSuitPosY*, on the grid are being incremented in the *y* direction and decremented in the *x* direction. From the grid, as the *y* value becomes more positive, you can see you are heading downward. As the *x* value becomes more negative you are moving to the left. Running the example values through the code you have the following:

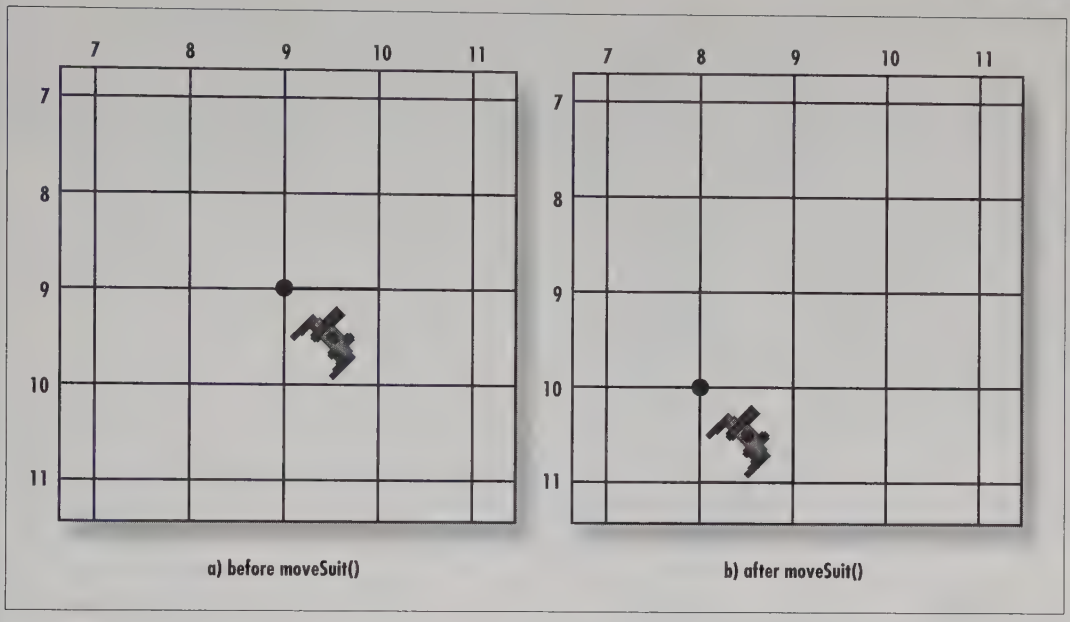


FIGURE 2-25 PowerSuit movement inside the logical grid

```
movement = 1;
pwrSuit->pwrSuitPosY += movement;
pwrSuit->pwrSuitPosY = 9 + 1 = 10
pwrSuit->pwrSuitPosX -= movement;
pwrSuit->pwrSuitPosX = 9 - 1 = 8
```

The new logical coordinates for the PowerSuit are at *pwrSuitPosX* = 8 and *pwrSuitPosY* = 10. The PowerSuit's new location is shown in Figure 2-25(b). The PowerSuit is bounded in the region of $0 \leq x \leq 19$ and $0 \leq y \leq 14$. The following code is used to prevent the PowerSuit from leaving these bounds:

```
#define MIN_GRID_X      0
#define MIN_GRID_Y      0
#define MAX_GRID_X      19
#define MAX_GRID_Y      14
// If powrsuit is not moving off the screen then let suit
// start moving.
if (pwrSuit->pwrSuitPosX >= MIN_GRID_X
     && pwrSuit->pwrSuitPosX <= MAX_GRID_X
     && pwrSuit->pwrSuitPosY >= MIN_GRID_Y
     && pwrSuit->pwrSuitPosY <= MAX_GRID_Y)
            pwrSuit->moveFrame = 1;
```

A check is made to determine if the PowerSuit is still in bounds. If it is, then the *moveFrame* field is set to one. This indicates that the PowerSuit may take one

step in the direction it is traveling. The PowerSuit in this particular strategic simulation is designed to make four steps per move. Another check is made to see if the PowerSuit stepped out of bounds. If it did, each specific case is handled to prevent the PowerSuit from stepping off the grid. This is handled with the following lines of code:

```
// Make sure the numbers stay within the screen bounds.
if(pwrSuit->pwrSuitPosX < MIN_GRID_X)
  pwrSuit->pwrSuitPosX = MIN_GRID_X;
if(pwrSuit->pwrSuitPosY < MIN_GRID_Y)
  pwrSuit->pwrSuitPosY = MIN_GRID_Y;
if(pwrSuit->pwrSuitPosX > MAX_GRID_X)
  pwrSuit->pwrSuitPosX = MAX_GRID_X;
if(pwrSuit->pwrSuitPosY >MAX_GRID_Y)
  pwrSuit->pwrSuitPosY = MAX_GRID_Y;
```

The PowerSuit is governed by its movement in the logical grid. The PowerSuit in the test, however, was displayed by mapping the logical grid coordinates to the DIB battleground. In this way, a simulated terrain mapping was added to the test to provide a degree of realism.



Rendering the PowerSuit

PowerSuit movement to this point has been described in terms of logical coordinates. The grid is an imaginary boundary smaller in scale than the actual offscreen DIB by a factor of 32 units per pixel, used to keep track of the PowerSuit's moves on a 2D plane. One step in the grid represents 8 pixels on the screen. Four steps are required by the PowerSuit to move from one unit grid to the next. Four steps in a unit grid are equivalent to 32 pixels on the offscreen DIB. As the progress of the PowerSuit was updated in function MoveSuit(), the *movex* and *movey* fields were updated as well with the values of 0 or ±8 pixels in the desired heading. This information is sufficient for the PowerSuit to map its coordinates back onto the offscreen DIB for running these initial simulations.

The mapping of grid coordinates back to physical DIB values requires four steps. These steps are shown in Figure 2-26. These steps are broken up into calculating RECTs, drawing sprites, determining an encompassing RECT, and updating the screen DC. The following is a discussion of the individual steps.



Step 1. Calculating Sprite RECTs from Logical Grid Coordinates

The Windows data type RECT is used for holding the coordinate mappings from the grid to the actual DIBs used for the animation sequences. The first

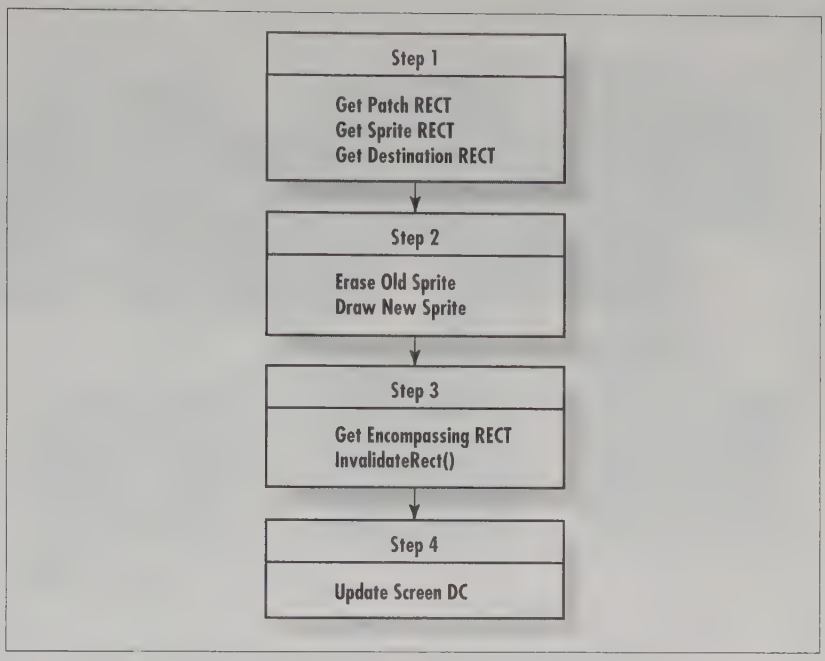


FIGURE 2-26

Steps for drawing the PowerSuit to the screen DC

RECT of importance is called a patch RECT. For boilerplate testing, Windows 95 animation is adequate and some measures for speed must be taken into account. To minimize the time required for copying the entire background DIB over the entire offscreen DIB for erasing old sprites, a patch RECT is provided to only erase the location of the sprite. The CPU time required by Windows 95 to blit a 32 x 32 pixel region is significantly faster than that of blitting the entire 640 x 480 pixel DIB for each screen DC update. Figure 2-27 shows the layout of the background DIB object, which contains the PowerSuit sprites and the master background image. The offscreen DIB object is used for rendering new frames for updating the screen Device Context.

Updates from the background DIB object to the offscreen DIB object are performed in cycles of calculating the patch RECT, for covering up the old frame, the calculation of the sprite RECT, the location of the sprite cell to be used on the background DIB, and its destination RECT, which is where the sprite will be drawn.



Calculating the Patch RECT

The value of the patch RECT is calculated from values stored in the *powerSuitInfo* structure. The fields *pwrSuitPatchX* and *pwrSuitPatchY* are updated from the fields *pwrSuitPosX* and *pwrSuitPosY* before the offscreen DIB is updated. The function to do this is as follows:

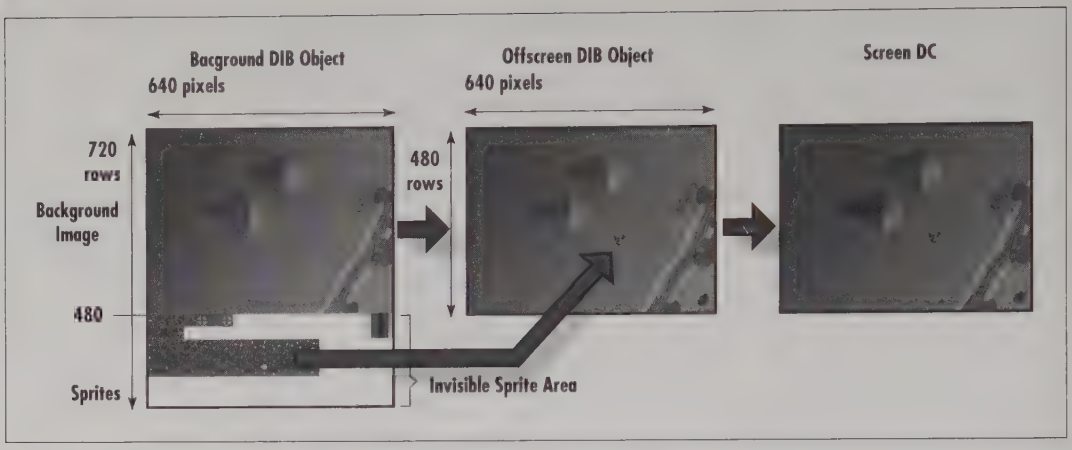


FIGURE 2-27 The background and offscreen DIB objects

```
void UpdatePatchRect()
{
       //Update the Patch points.
        goodPowerSuit.pwrSuitPatchX = goodPowerSuit.pwrSuitPosX;
        goodPowerSuit.pwrSuitPatchY = goodPowerSuit.pwrSuitPosY;
}
```

By updating after each offscreen update, the *pwrSuitPatchX* and *pwrSuitPatchY* values will always contain the top-left coordinates of where the PowerSuit was on the offscreen DIB. When the *pwrSuitPosX* and *pwrSuitPosY* values get updated, the previous position of the PowerSuit is kept in the *pwrSuitPatchX* and *pwrSuitPatchY* values. These values are shown in Figure 2-28(a) and 2-28(b).

With each move in the simulation, the PowerSuit takes four steps. Each step represents a movement of 0 or ±8 pixels on the screen. The size of the patch required for completely covering the *PowerSuits* movement is calculated as follows from the values generated by MoveSuit():

```
RECT patch;
// Find the background area that needs to be mended.
setGameRect(&patch,
             pwrSuit->pwrSuitPatchY*SPRITE_SIZE
                    + (pwrSuit->movey*(pwrSuit->moveFrame-1)),
             pwrSuit->pwrSuitPatchX*SPRITE_SIZE
                    + (pwrSuit->movex*(pwrSuit->moveFrame-1)) );
```

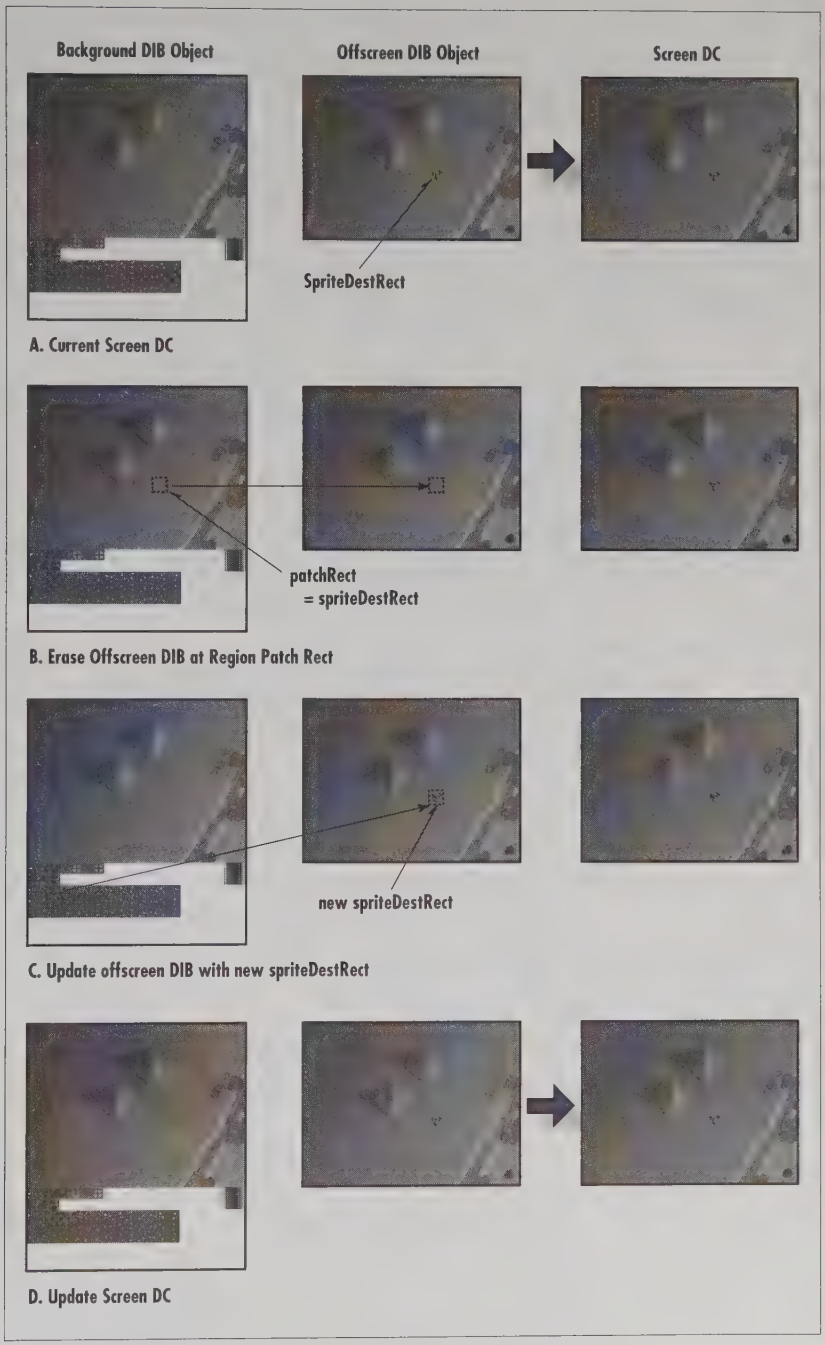


FIGURE 2-28
Handling a sprite update to the screen DC

The function setGameRect() is used to set a RECT based on passing in arguments of the top and the left coordinates of the RECT. The RECTs used in the simulation are typically bottom = top + 32, and right = left + 32. The code for setGameRect() is as follows:

```
#define SPRITE_SIZE 32
void setGameRect( RECT *r, int top, int left )
{
  r->top = top;
  r->left = left;
  r->bottom = r->top + SPRITE_SIZE;
  r->right = r->left + SPRITE_SIZE;
}
```

For the patch RECT, the *patch.top* coordinate is calculated by getting the previous PowerSuit coordinate at *pwrSuitPatchY* * 32. This maps the *lprevious* logical grid coordinate over to the 640 x 480 offscreen DIB. If the PowerSuit were stationary, you would be sitting right on top of where the DIB needs to be mended. However, it can take up to four steps of eight pixel steps each. This is the maximum vertical distance the PowerSuit could have covered if it did take four steps. This offset must be included in the computation. The top value of the patch RECT is then as follows:

```
top = pwrSuit->pwrSuitPatchY*SPRITE_SIZE
            + (pwrSuit->movey*(pwrSuit->moveFrame-1)),
```

The same computations are required to determine the left value of the patch RECT:

```
left = pwrSuit->pwrSuitPatchX*SPRITE_SIZE
            + (pwrSuit->movex*(pwrSuit->moveFrame-1)) );
```

Precisely the same computations are performed to determine where the new sprite destination RECT is on the offscreen DIB. This is the location the PowerSuit will be drawn after the patch RECT erases the old position on the DIB. The top and left coordinate values of the sprite destination RECT are calculated as follows:

```
top = pwrSuit->pwrSuitPatchY*SPRITE_SIZE
            + (pwrSuit->movey*pwrSuit->moveFrame),
left = pwrSuit->pwrSuitPatchX*SPRITE_SIZE
            + (pwrSuit->movex*pwrSuit->moveFrame) );
setGameRect( &destination, top, left);
```

This computation represents the region pointed to in Figure 2-28(c), showing the new *spriteDestRect* in the offscreen DIB. One final RECT computation remains to be performed, the calculation of the source sprite cell on the background DIB. Figure 2-29 represents the storage orientation of the PowerSuit sprites on the background DIB. This figure also shows the computations of the base sprite cell and the offsets of the other sprites from this one at location 0,480 in the DIB. The desired sprite cell is retrieved with the function getSpriteRect(). Its code looks like this:




```
void getSpriteRect( RECT *r, pwrSuitInfo *pwrSuit)
{
  r->top = pwrSuit->spriteRect.top + (pwrSuit->heading * SPRITE_SIZE);
  r->left = pwrSuit->spriteRect.left + (pwrSuit->spriteFrame * SPRITE_SIZE);
  if (pwrSuit->firingWeapon)
    r->left += 96;
  r->bottom = r->top + SPRITE_SIZE;
  r->right = r->left + SPRITE_SIZE;
}
```

Notice the sprites starting at cell (0, 544) in Figure 2-29 are stored column-wise in respect to their heading. The sprite cell's top coordinate can be calculated using this column indexing scheme:

```
top = pwrSuit->spriteRect.top +  (pwrSuit->heading * SPRITE_SIZE);
```

The left coordinate of the cell can be calculated based on which sprite frame is desired from the DIB:

```
left = pwrSuit->spriteRect.left + (pwrSuit->spriteFrame * SPRITE_SIZE);
```

These are all the RECT values that are required for updating the offscreen DIB. The next step in the process is to erase the offscreen DIB patch and draw the PowerSuit sprite to its new location.



Step 2. Patching Backgrounds and Rendering Sprites

This is the simplest step in the exercise of drawing to the offscreen DIB. All the values required were previously calculated in Step 1. The only work required here is to erase the old PowerSuit sprite on the offscreen DIB:

```
pDibOffscreen->CopyDIBBits( *pDibBckGrnd, patch, patch );
```

and draw the sprite to the new destination on the DIB:

```
pDibOffscreen->CopyTransDIBBits( *pDibBckGrnd,
                    drawRect, destination, transParentPixel );
```

The complexity of DIB blitting is in the setup. Two calls to the CDib member functions are all that is necessary for the patching and rendering. The offscreen DIB is now ready to update the screen DC.



Step 3. Sending a WM_PAINT Message

The Windows 95 operating system, for GUI based applications, is message-based. The screen DC does not update unless a WM_PAINT message is processed from the message queue. A message can be sent to the operating system by invalidating a region of the client window. One method is to invalidate the entire client window to force a refresh. However, if only a specific region of the client window is invalidated, then an entire window refresh is not

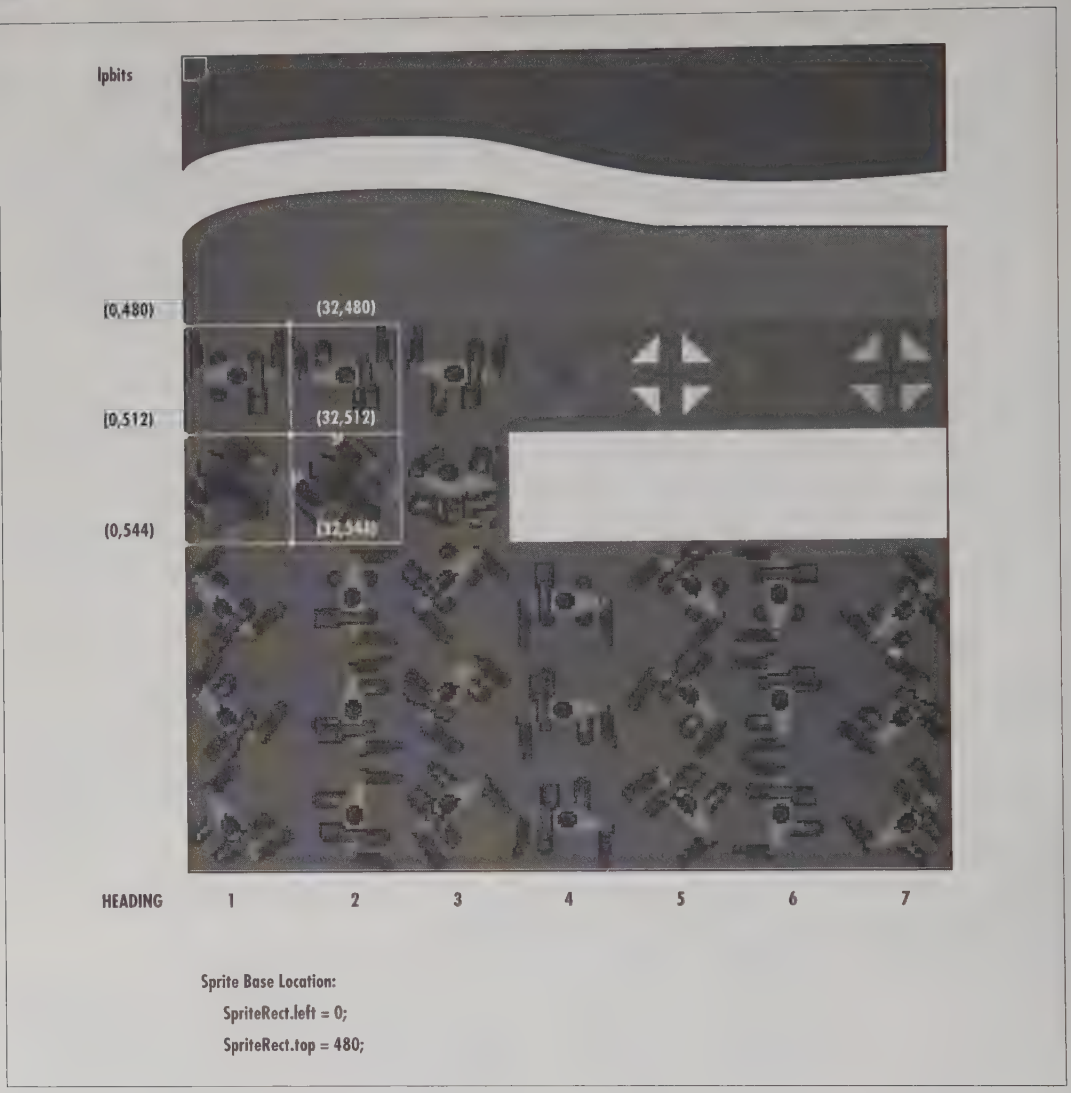


FIGURE 2-29 Sprite orientation in background DIB

necessary. This requires calculating a RECT that encompasses where the patch RECT erased and the destination RECT of where the sprite was just drawn. An encompassing RECT is shown in Figure 2.30. The encompassing RECT is calculated with the function getEncomapssingRect(). The code looks like this:

```
void getEncompassingRect( RECT *patch, RECT *encompass)
{
  if (patch->top < encompass->top)
    encompass->top = patch->top;
```

```
  if (patch->bottom > encompass->bottom)
    encompass->bottom = patch->bottom;
  if (patch->left < encompass->left)
    encompass->left = patch->left;
  if (patch->right > encompass->right)
    encompass->right = patch->right;
}
```

The encompassing RECT, as shown in Figure 2-29, is the smallest RECT that contains all the offscreen DIB changes. A message is sent to the operating system with the Windows 95 API call InvalidateRect(), using these parameters:

```
InvalidateRect(hwnd, &encompass, FALSE);
```

Concluding this call, a message is put in the queue for WM_PAINT to repaint the region of the client window. The WndFunc() CALLBACK function is now responsible for taking care of the updates.



Step 4. Updating the Screen DC

The Windows 95 API routine InvalidateRect sent the message WM_PAINT to refresh a region of the client window. The WndFunc() must be given functions to handle this update. The WM_PAINT handler follows:

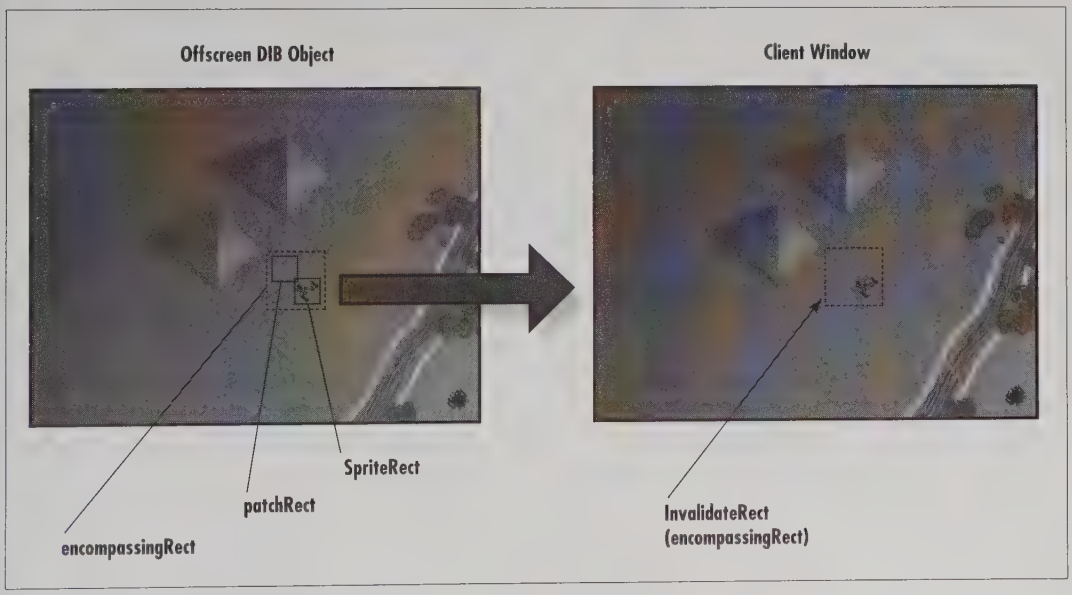


FIGURE 2-30 Increasing screen DC updates using InvalidateRect()

```
case WM_PAINT:
  hdc = BeginPaint (hwnd, &ps);
  RefreshScreen(hwnd, hdc);
  EndPaint (hwnd, &ps);
  break;
```

The case statement is almost standard with the exception of the function RefreshScreen(). This is the routine provided by the PowerSuit application to handle the screen DC updates. The code for RefreshScreen() is as follows:

```
void RefreshScreen(HWND hwnd, HDC hdc)
{
   HPALETTE hPalette = pDibBckGrnd->CreateLogicalPalette();
   HPALETTE hOldPalette;
   SelectPalette(hdc, hPalette, FALSE);
   RealizePalette(hdc);
   pDibOffscreen->DrawBits(hdc,0,0);
   SelectPalette(hdc, hOldPalette, FALSE);
   DeleteObject(hPalette);
}
```

The function RefreshScreen() retrieves the palette used by the DIB object and selects this palette when it updates the DC. The CDib member function, DrawBits(), is called by the offscreen DIB object to update the DC. When the update has finished, the function RefreshScreen() returns the original values of the palette and deletes the handle to the palette used to retrieve the CDib object's palette. The screen DC is now updated.



Findings and Remarks

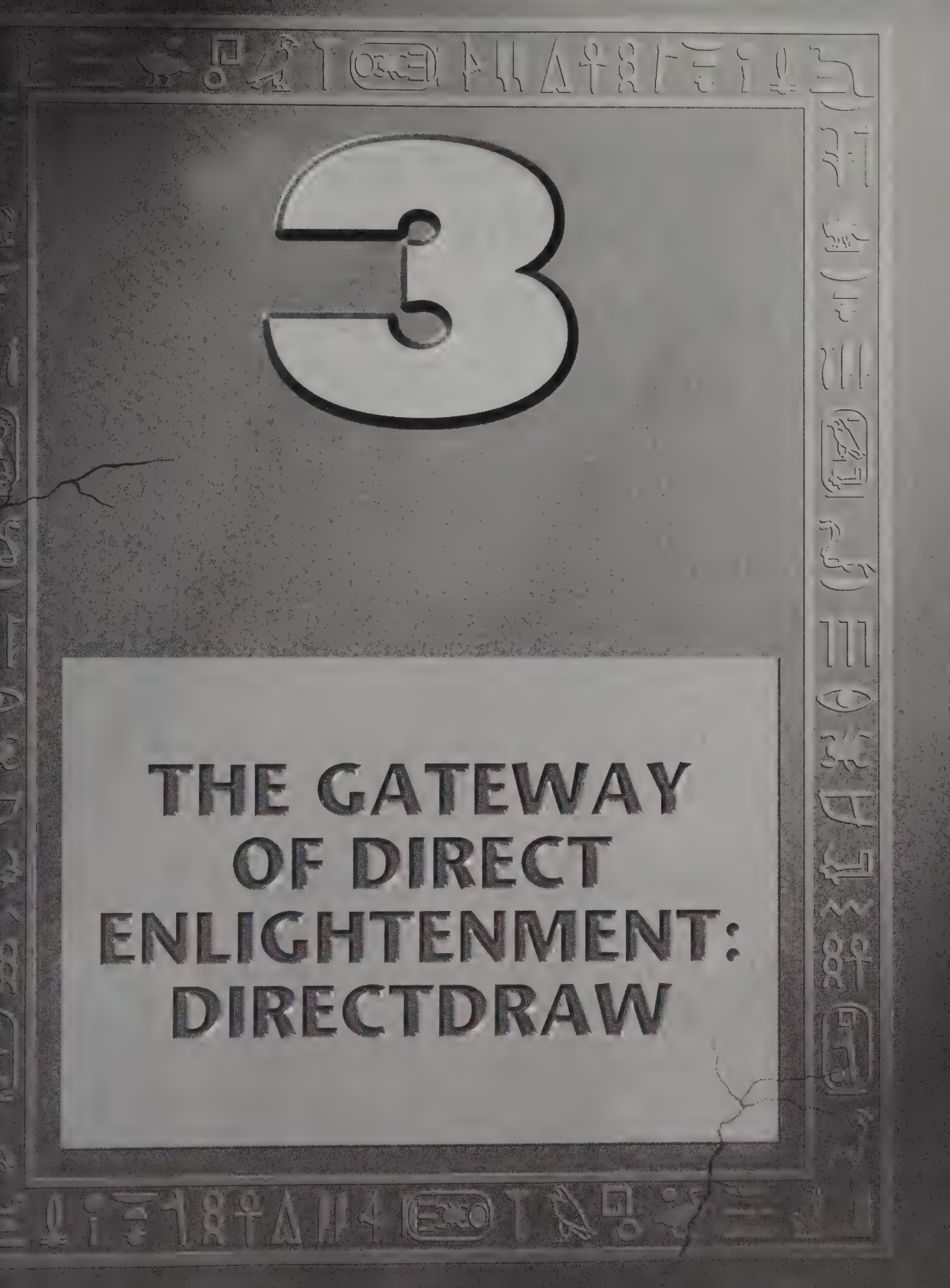
The PowerSuit test using the Windows 95 operating system was sufficient for the boilerplate design. However, the operating system and its methodology in performing DC updates lacked the performance features of a DOS environment where the screen memory can perform page flipping. One suggestion is to modify the existing test code and use the Windows 95 Game SDK DirectDraw API. Since the DirectDraw API uses Device-Independent Bitmaps as its primary format for porting images into its memory surfaces, this approach may take the least amount of time for setting up a second PowerSuit test. It is strongly recommended that the PowerSuit be ported over to the Windows 95 Game SDK APIs for real-time testing.





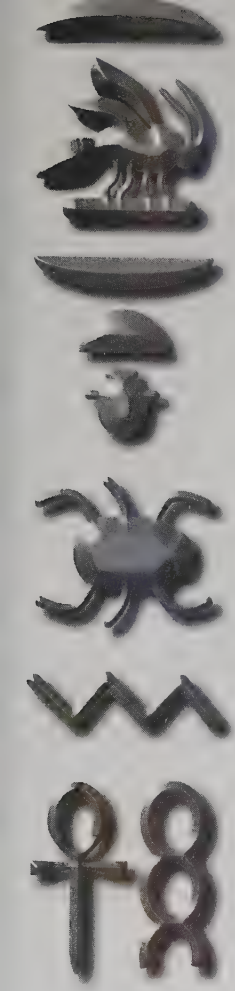
Conclusion

It's amazing how much text is needed to cover the manipulation of Device-Independent Bitmaps. I know the Windows 95 Game SDK provided simple tools and examples for reading in a DIB from a .BMP file. However, as I pointed out earlier, the examples used things such as color keys whose values are 0x00 (all black). Artists use black for shading and gradations of light. The CDib object provides tools like these to provide flexibility and a starter kit to set you on your way with similar tools like you had when you worked with IFF and PCX files. These tools include ways to get at palette values, retrieve individual color values from DIBs, and set pixels on DIBs. There is a mountain of code here to use and modify to your own discretion. As a personal side note, I was thinking about using inheritance features of C++ to write a class that reads IFF files and translates them into a DIB object. Using a powerful graphics program such as *Photoshop*, I can create my game art on the Macintosh and save it in virtually any format I want. But I did find that just working with DIBs in *Photoshop* was the easy way to create game art.



3

THE GATEWAY OF DIRECT ENLIGHTENMENT: DIRECTDRAW



THE GATEWAY OF DIRECT ENLIGHTENMENT: DIRECTDRAW

The previous chapter examined the Windows 95 supported graphics file format, the Device Independent Bitmap. You created a DIB object for reading a DIB from a .BMP file and manipulated its image and palette bits. You found the DIB was highly suited for graphics composition but unsuited for video game-quality animation. This chapter takes a look at the features of DOS video performance capabilities that made it such an attractive gaming platform over the past decade and how the DirectDraw development team addressed these issues to make Windows 95 the platform of choice for the next generation of video games.

EPISODE THROWING A WRENCH INTO THE PLAN



A crew of technicians worked feverishly to get some answers as to why the Moebius reactor suddenly came online. Overlooking the operation was a man as formidable as the Queen of Hearts in *Alice in Wonderland*. He stood by and watched his subordinates. He calmly blew rings of smoke into the air as he waited for their results. The man stood silently, deep in thought, equating, planning, and scheming. Bringing Maelstrom into the project was a mistake. A mistake he would hoped he wouldn't soon regret.

The man blew another ring of smoke into the air. He knew why this device of alien origin suddenly came alive after nearly forty years. There was only one creature around capable of understanding the quantum physics of this reactor.

Only one person in this complex who contained strains of alien DNA. And that one person, that little brat, was nowhere to be found.

"Forget your damn measurements!" the dark-suited man said as he threw down the butt of his spent cigarette. "I know it's online. I want to know where the energy is being fed to."

"But Sir, we don't know who or how or why the device has suddenly become operational."

The cigarette-stenched man took a step towards his technician. "I know the who and the why and I know the how. I want to know the where, and I want to know now! If you can't provide me with those results I'll get someone down here who can."

The man's soulless glassy eyes made contact with the technician's fearful eyes. The technician froze, drawing on every ounce of energy to appear as if he wasn't scared.

"I'll need access to documents and blueprints as far back as the late 1940s," the shaken man informed.

"You won't need those," the dark-suited man commented. "I was here when they boxed in that abomination and the gray visitors left our world. The Moebius reactor is entombed. Now find out how somebody tapped its energy, and find out now."

The sinister man turned and walked away. He knew what was going on. 13 brought the Moebius reactor online to build a virtual reality gate. He had read the papers and journals of 13. The PowerSuit project required tremendous amounts of energy. 13 and I were going into the artifact. Our intentions were no doubt to destroy Agamemnon. This would ruin everything.

13 had mentioned in his papers a computer of unknown origin. This would have to be a substantial piece of hardware. But where did he hide it? This complex was full of underground labyrinths and blocked-off mazes. Some of these underground tunnels predated his term of rule down here. No telling what his predecessors hid in these mazes or how deadly these tombs of forbidden technology would be.

He knew he would have to search these tunnels to put an end to 13 and my meddlings. 13 and Maelstrom were about to ruin everything. The artifact had the power to dominate the world's information processing as we knew it. With information comes power, the power to control governments, currencies, and races of people. This couldn't fall apart now.



PC Video Architecture

A game like *Tetris* doesn't require speed for blitting nor does it necessitate entire screen updates. Simple puzzle games such as these can get away with low frame rates in the range of 15 frames per second (fps), as would be the case for a DIB style game. It was even demonstrated that for a simple pure Windows 95

DIB-based game, *PowerSuit*, which relies solely on the GDI, this is acceptable. However, the game *Tetris* and 13's DIB *PowerSuit* test program have limited dimensions of movement for the gameplay. The backgrounds are static. In *Tetris*, it's a backdrop showing the Czar's palace. In the *PowerSuit* application, you have a fixed terrain that doesn't scroll. There are no external dependencies on what kind of hardware you have. The blitting is being triggered by a user action to update the screen. A message is posted to your game application causing the application to interact with the GDI. Figure 3-1 shows the different abstraction layers that are used by a Win32 DIB-based game.

Some developers for Windows game applications have diligently squeezed out frame-rates comparable to those in DOS, 20-35 fps. A good example of such an application is Activision's port of *Pitfall*. However, the downside to *Pitfall* was the number of independent video drivers Activision had to code to get these results across multiple PC configurations. Pulling this off in a Windows environment is no easy accomplishment.

DOS, on the other hand, is highly suited for game programming because of its real mode architecture. Programmers can take full control of the PC and load in any drivers they feel suited to run their game applications. In later years, protected-mode software development kits (SDKs) were introduced to help programmers grab more system resources that were required for the more complicated games that were emerging, like *DOOM*. The quest for more memory, faster processors, and full 32-bit addressing pushed the applications onto DOS extenders, like DOS4/GW. Extenders are mini operating systems that load in over DOS to create a protected mode environment. Protected mode or ring protection is depicted in Figure 3-2. It is an onion-skin-layered operating system where different tasks are assigned to their layers by a protection level number.

The innermost layer belongs to the operating system kernel and its tasks have the highest priorities in the system. There are intermediate layers for

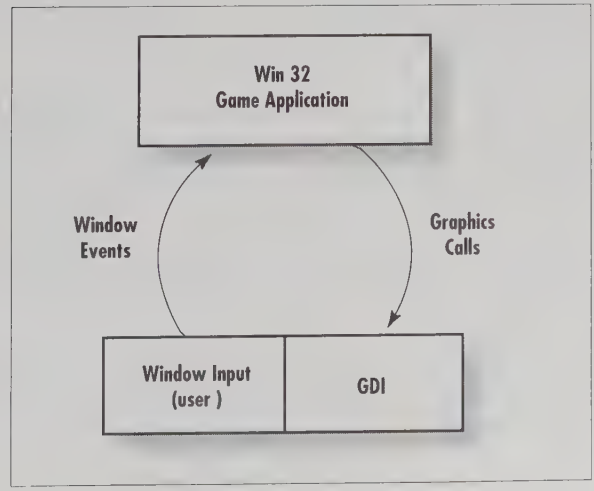


FIGURE 3-1
Overview of a Win32 game application

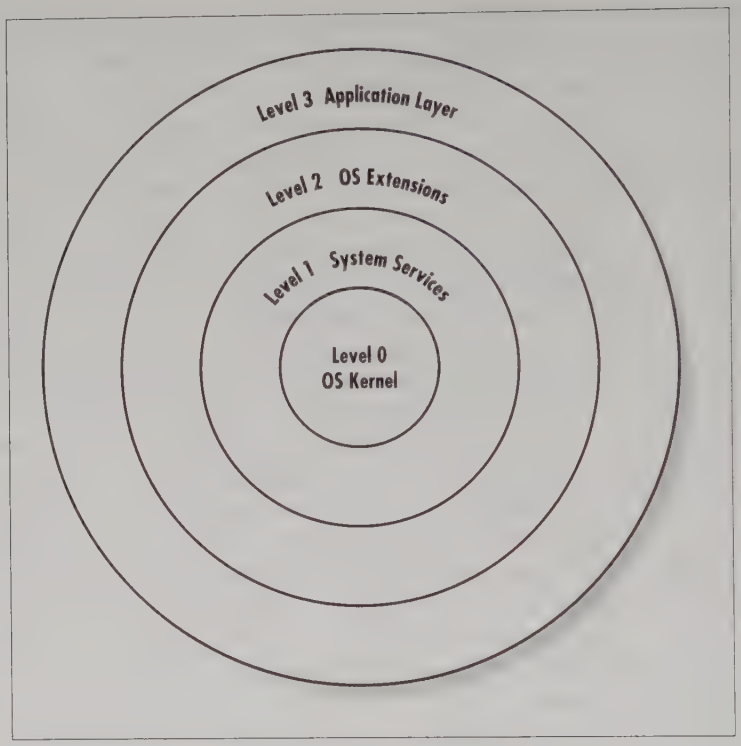


FIGURE 3-2
DOS extender ring protection layers

things such as disk access and so on to the outermost layers, which are your applications (i.e., your video games). The layering is protected because your application or task cannot access resources in use by the kernel layer. The concept is that a task can access resources higher up in the layering, but it cannot access resources further down. How does this differ from real mode DOS? Imagine one task, your game, running at the same layer as your kernel. In real mode the game application can take system resources and crash the PC. In protected mode, an application-level task attempting to access system resources generates a general protection fault error.

The Intel 286 processor was capable of 24-bit protected mode addressing, but the 386 actually revolutionized protected mode application design using 32-bit linear addressing. Linear memory is also sometimes called flat addressing. The benefits to the game designers are that they can allocate linear memory chunks for their games in sizes of megabytes rather than designing memory management schemes around multiple DOS chunks of 64K. DirectDraw is a 32-bit API that takes full advantage of this flat addressing scheme.

When Windows 3.0 was introduced, it was actually a 16-bit protected mode DOS extender, but it didn't have any of the benefits of linear memory and flat addressing. Its video requirements, as you have seen, are limited to the capabilities of the GDI layer. Layer in this context refers to the software abstraction in Figure 3-1 and not a protected mode layer. Rather than hacking

and chopping at the Windows 3.1 operating system, game programmers found it easier to use DOS and a DOS extender SDK. The Watcom compiler shipped with the DOS4/GW 32-bit extender soon became the game industry choice. The DOS4/GW, like other DOS extender SDKs, provided protected mode hooks to lower layers of the operating system. This meant programmers could tweak and poke at PC ports to their hearts' content as long as their software was well-behaved. The greatest benefit from these extenders was 32-bit writes to video memory. The Windows environment was closed to lower layer access, especially to ports.

When Windows 3.0 shipped, it supported 16 colors on the VGA. Video games of that time required 256 colors on EGA or VGA graphics adapters. Through the Windows evolution, no routines or calls were introduced to assist low-level programmers, such as those writing video drivers for games, to have access ports and video hardware. The Windows device driver development kits (DDKs) were more geared toward hardware vendors and their products. So, for a long time, dating back to Windows 3.1's introduction, the Microsoft Windows operating system was regarded as not well suited for game development. Professional game developers need access to and control of hardware to provide the best performance of visual and audio effects that are important in games.

Ironically, during this period, video card vendors were providing hardware blitters in their chip sets for Windows acceleration that were capable of outperforming anything game consoles such as Nintendo and Sega were capable of. One of the stories I have heard about how the Windows 95 Game SDK came to be was that two software engineers, who just so happen to be the two prominent figures in the project, were working out with weights in a gym, discussing the capabilities of current hardware and how they could be utilized for video games. One of the programmers worked on the WinG API and knew its limitations. The WinG interface was unsuited for commercial game applications. His insight from WinG led him to design the architecture, shown in Figure 3-3, on the back of his napkin during a power lunch at Burger King.

The new game architecture involved bypassing the GDI and utilizing software hooks to communicate between the Windows 95 operating system and the physical hardware layer. This marriage provided hardware capabilities to game designers that were previously only available to, and *unused* by, Windows applications. Windows application developers didn't utilize graphics hardware accelerators because spreadsheets and word processors didn't require this performance. The GDI interface was sufficient. An interesting side note is that hardware vendors were removing the special chip sets from their designs because nobody was utilizing these features in their commercial software applications. This vision that began in a gym soon inspired video hardware developers to focus on providing video drivers that would snap into the Windows 95 operating system and utilize the unique features of their product. From the programmer's point of view, access is made seamlessly via

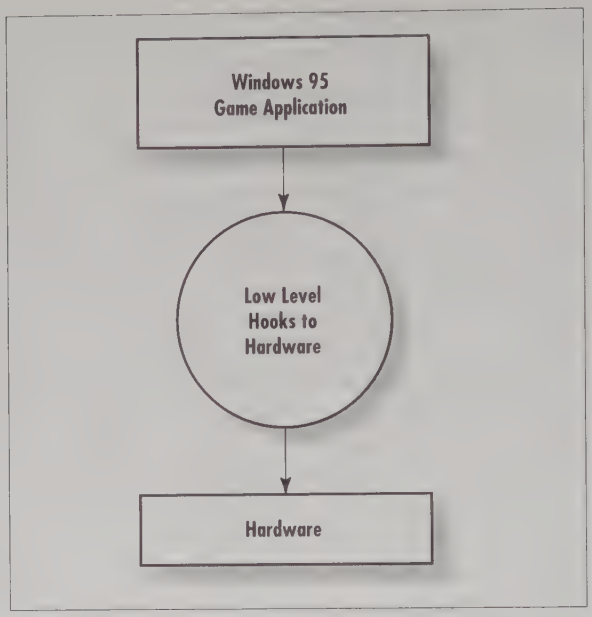


FIGURE 3-3
Game architecture for Windows 95

the API. The call to an S3 chip set for blitting is the same as a call to an nVidia or Mach 64 chip set. The operating system and the intelligent graphics interface called DirectDraw would handle and negotiate the underlying hardware needs and dependencies.



DOS Video Architecture

Having DOS programmers switch over to the Windows 95 operating system was going to be a very tough sale. Microsoft Windows was regarded as an office desktop for spreadsheets and word processing. The WinG interface was extremely underpowered in terms of what professional game developers required to move over to the Windows operating system. WinG only offered enhancements to handling palettes and blitting DIBs to the screen DC. No significant improvements to increasing frame-rates were introduced. The biggest problem still remained: no hooks to the lower hardware layers.



BIOS-Supported Video Mode 13h

PC game designers use primarily one of two video modes in their games. One of those modes is 13h, a 320 x 200 pixel with 256 colors. Visual C++ 32-bit development systems do not provide hooks to DOS. Most game developers have Watcom development systems on their PCs as well as Microsoft Visual

C++. I will provide 32-bit DOS programming examples using the Watcom compiler. These examples can easily be ported over to the Visual 1.5 16-bit DOS development system. An example of setting Mode 13h using a 32-bit DOS extender is provided in Listing 3-1.

LISTING 3-1 Setting the video to Mode 13h using DOS4/GW

```
int videoMode = 0x13;
union REGS regs;

regs.w.ax = videoMode;
int386(0x10, &regs, &regs);
```

What makes Mode 13h really popular with developers and hobbyists is that its video memory configuration is four contiguous planes of video memory. To fully appreciate the task ahead of the Direct X Game SDK Development team, you're going to have to brush up on some of your video hardware architecture. The standard VGA card you buy at your computer store for $39.95 (the 256 color Bargain Mart Special card) has four banks of memory, as shown in Figure 3-4. Each bank of memory is 64K in size and is called a plane of memory. Remember from Chapter 2 that a DIB can have multiple planes? Well, this is where the planes come into play.

In video Mode 13h, all four planes are chained so the pixels are located in memory in a linear sequence. In this mode, for each pixel, four planes of memory are contiguous. The disadvantage of a linear memory scheme is that this throws off the pixel spacing, making them non-square pixels. The concept of chaining planes for video Mode 13h is shown in Figure 3-5.

Mode 13h is the only linear video mode supported by BIOS that has 256 colors. Accessing this memory is simple too. You set a pointer to the start of the VGA memory and offset your way into each linear address space. The code for this using Watcom C++ is as follows:

```
char *pVGAPtr = (char*)(0xA0000);
```

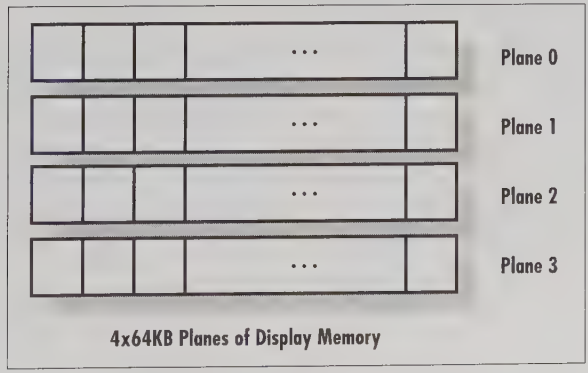


FIGURE 3-4
The four memory planes of a VGA card

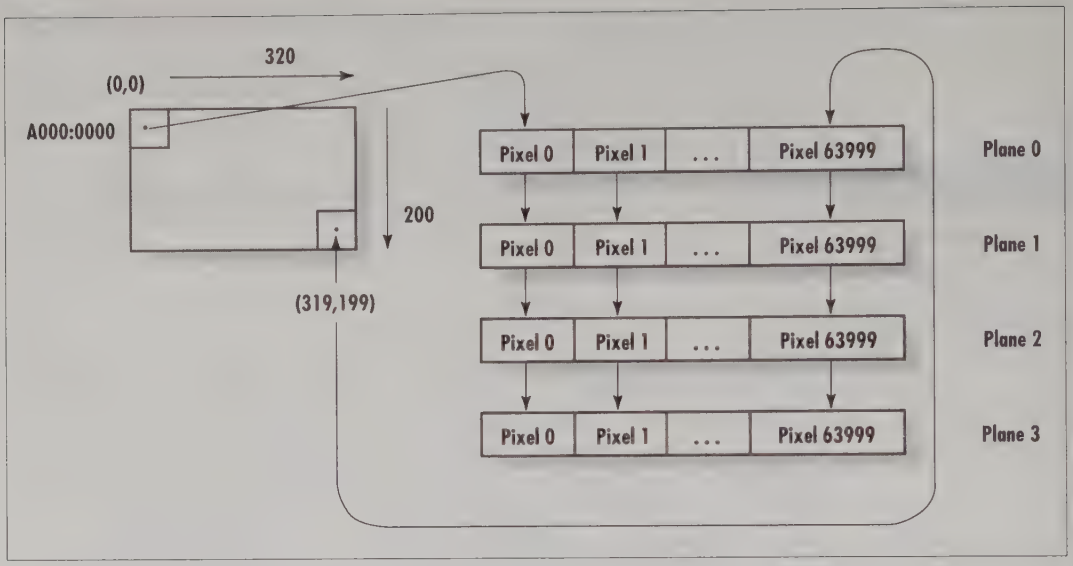


FIGURE 3-5 Chained video planes in Mode 13h

The BIOS-supported video mode is popular among shareware developers because of its ease of use. Hybrid modes of 13h have been designed to support page flipping. I was pretty much a Mode X advocate until I paid a visit to my brother-in-law's company, Sculptured Soft, where they were working on *Mortal Kombat 3* for the PC. This game uses hybrid Mode 13h and supports page flipping. By hybrid I mean you have to tweak a lot of VGA registers, mode 13h page flipping is not directly supported by the BIOS. Based on the threats that they would alter my gender with blunt instruments, I can't disclose how they did this.

Video Mode X, the Game Developer's Secret

A popular DOS video mode in use by commercial developers is Mode X. This is a super hybrid Mode 13h that supports page flipping. The memory configuration of Mode X is a plane memory organization, unlike Mode 13h, which is chained linear. Mode X is a 320 x 240 pixel mode with 256 colors. Because it is plane memory, this scheme can take full advantage of the video card hardware and is extremely fast. To give you an idea of an application for this, *DOOM* is using Mode X. Figure 3-6 shows the Mode X plane memory configuration and its pixel orientation.

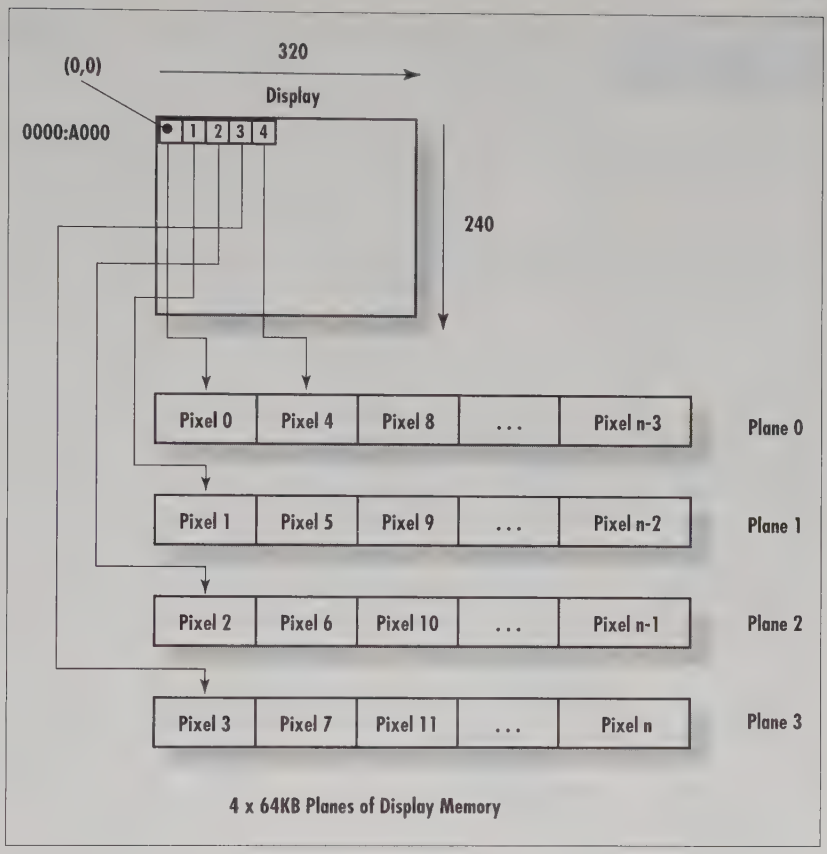


FIGURE 3-6
VGA Mode X plane memory organization

From the figure you see that Mode X is one plane per pixel, where Mode 13h is four planes per pixel. This video mode with a resolution of 320 x 240, with square pixel spacings, has an aspect ratio of 1:1. Configuring Mode X was pretty much an industrial secret for quite some time until Michael Abrash published detailed articles in *Doctor Dobb's Journal* and his book, *Zen of Graphics Programming*. 13 was kind enough to give you the Watcom C++ code he ported over from Abrash's articles that were all written using Microsoft Assembler. As I mentioned, Mode X is a super hybrid Mode 13h. You begin setting Mode X by setting Mode 13h as you did previously; however, you must perform some low-level surgery and tweak the video card's CRT controller (CRTC) and sequence controller (SEQC) registers to get your desired mode. The code used by 13, to set Mode X, is provided in Listing 3-2.

LISTING 3-2 Setting Mode X using a DOS4/GW 32-bit DOS extender

```
// ports to registers on VGA card
#define SEQC 0x03c4     // port number for the sequence controller
#define CRTC 0x03d4     // port number for the CRT controller

// definitions of port values for Mode X
// CRT Controller registers and corresponding values
#define VERTICAL_TOTAL          0x00d06
#define OVERFLOW                0x03e07
#define MAX_SCAN_LINE           0x04109
#define VERT_RETRACE_START      0x0ea10
#define VERT_RETRACE_LOW        0x0ac11
#define VERT_DISPLAY_END        0x0df12
#define UNDERLINE_LOCATION      0x00014
#define VERT_BLANK_START        0x0e715
#define VERT_BLANK_END          0x00616
#define MODE_CONTROL            0x0e317

void setModeX()
{
  int videoMode = 0x13;
  union REGS regs;

  regs.w.ax = videoMode;
  int386(0x10, &regs, &regs);

  outpw( SEQC, 0x0604 );
  outpw( SEQC, 0x0100 );
  outp( MISC_OUTPUT, 0xe3 );
  outpw( SEQC, 0x0300 );
  outp(CRTC, 0x11 );
  outp(CRTC + 1, (inp(CRTC + 1) & 0x7f));
  outpw( CRTC, VERTICAL_TOTAL );      //  vertical total
  outpw( CRTC, OVERFLOW );            // overflow, bit 8 of vert
  outpw( CRTC, MAX_SCAN_LINE );       // set for double scan
  outpw( CRTC, VERT_RETRACE_START );  // vsync start
  outpw( CRTC, VERT_RETRACE_LOW );    // v sync end and protect
  outpw( CRTC, VERT_DISPLAY_END );    // vert displayed
  outpw( CRTC, UNDERLINE_LOCATION );  // turn off dword mode
  outpw( CRTC, VERT_BLANK_START );    // v vblank start
  outpw( CRTC, VERT_BLANK_END );      // v blank end
  outpw( CRTC, MODE_CONTROL );        // turn on byte mode
  outpw( SEQC, 0x0f02 );              // enable writes on all four planes
}
```

This code selects Mode 13h, then modifies the values in the CRT controller registers to select 480 scanlines (Vertical Total register). You then configure the Maximum Scanlines register to double-scan each line. This modification gives



you 240 scanlines in the vertical direction. Why didn't the developers just keep the better vertical resolution of 480? The amount of memory required for this configuration makes page flipping on the standard VGA card a physical constraint. However, two pages of 320 x 240 pixels with 256-color mode and some spare video RAM sit nicely in a standard video card. These attributes are what make Mode X popular to developers.

Programming with Mode X, as you can guess, is not as trivial as Mode 13h. Accessing Mode X video memory requires an understanding of how video memory works. To plot the first pixel in Mode X (using DOS4/GW), you can use your code for Mode 13h:

```
char *pVGAPtr = (char*)(0xA0000);
```

However, the Mode 13h memory was a chained linear configuration allowing an addressing scheme of 200 consecutive scanlines 320 bytes in length. Mode X addresses its memory directly on one of four individual video memory planes. Your horizontal resolution is 320 pixels. Divide this evenly across four planes: you have each plane containing 80 bytes per scanline. Your four planes are consecutive. For programming purposes, number them zero through three (0-3). You can determine which plane you are supposed to write to by performing a logical AND operation on the *x* coordinate you wish to plot to. This will require two steps to determine the plane you wish to perform a write operation to. The first step you must take is to bit filter the low-order 8 bits of your 32-bit integer, like this:

```
#define BIT_FILTER_MASK 0xff      // 8 bit filter mask (255)

int whichPlane, lowOrderBits;

xcoord = 156;
lowOrderBits = xcoord & BIT_FILTER_MASK;
```

This code provides you with the low-order 8 bits of the 32-bit *x* coordinate value. What is the significance of this? Well, remember your goal is to determine which memory plane to perform the write operation to. You have four memory planes that you have labeled 0 through 3. So, for your second step, you need to perform a logical AND operation on the two lowest order bits in *lowOrderBits*. The resulting value will be which plane in memory you need to write to. The operation looks like this:

```
#define PLANE_MASK 0x03      // examine 2 lowest order bits
 whichPlane = lowOrderBits & PLANE_MASK;
```

You now have the desired plane you need to issue a write operation to. You need to inform the sequence controller that you want to write to this plane. Listing 3-3 shows the code to perform a write operation to a video memory plane.

LISTING 3-3 Using DOS4/GW to perform a write to a video plane location

```
#define MAP_MASK            0x02
#define INITIAL_PLANE_BIT   0x100
#define PLANE_INIT_MASK     INITIAL_PLANE_BIT + MAP_MASK

regs.w.ax = PLANE_INIT_MASK
regs.h.ah = regs.h.ah << whichPlane;
outpw(SEQC, regs.w.ax);
```

This snippet of code sets the write registers to allow a write operation to your desired plane of memory. All you have to do now is determine which byte on this plane of memory you are going to write to. The code in Listing 3-4 shows how to compute a scanline offset from your video pointer *pVGAPtr*.

LISTING 3-4 Computing a scanline offset from *pVGAPtr*

```
#define BYTES_PER_PLANE 80
char pixelData;
unsigned int scanLineOffset = (y * BYTES_PER_PLANE) + ( x >>2);

pVGAPtr = (char*)(0xA000 + scanLineOffset); //select the location in
                                            //the plane
*pVGAPtr = pixelData; // write the data to the memory location
```

As you can see, there is a lot of overhead in setting video Mode X and writing a pixel to its video memory. If you've never worked with Mode X before you're probably wondering why even bother when Mode 13h is so simple to use in the first place? Mode X offers frame-rate capabilities not available with the BIOS-supported Mode 13h. This speed is achieved through a technique called page flipping.



Page Flipping

One problem that's immediately noticeable when programming animation sequences with Mode 13h is the screen flicker. The standard methodology to working with Mode 13h is to use an offscreen buffer 320 x 200 bytes in size in system memory. The rendering and manipulation of sprites is carried out in this buffer and then blitted to VGA memory starting at location 0xA000. Figure 3-7 shows your screen update scheme using this technique.

Shuffling system memory across the PC bus to video RAM is slow. Assembly language code can speed the transfer time up significantly, but there is an upper limit on the capabilities of this mode. Early attempts at this problem included writing code based on waiting for vertical retraces to occur. Later methodologies used cache writes on a 486 or better processor, and provided some improvement. However, the problem of screen flicker still exists in Mode 13h.

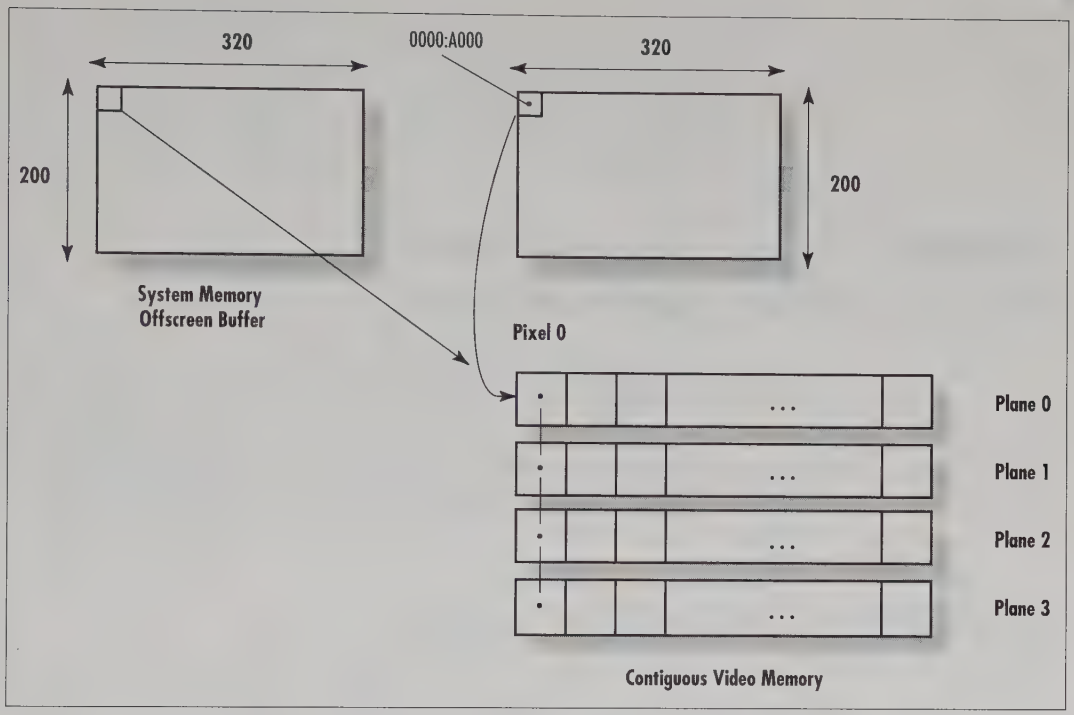


FIGURE 3-7 Screen updates with Mode 13h

Using page flipping, one buffer of video memory can be displayed while a secondary buffer is updated in the background. Since the user never sees the screen updates, the animation is smooth. When the secondary buffer is completed, it will be displayed while the previous screen buffer is being updated. When rendering has completed on that buffer it will be displayed to the screen. The user sees updates in as little time as it takes to update a page of video memory.

The downside to page flipping is properly coding the timing mechanism for flipping screens. The trick of waiting for vertical retrace, as you would for Mode 13h, is no longer an option. You are now dealing with hardware timing and latches on the VGA card. By the time your development process has reached this level, you are sitting in your development lab with an SVGA reference guide, looking at compatibility and timing issues. Usually, it's the inexpensive Bargain Mart VGA cards that are giving you your biggest headaches. One of the most unorthodox methods I have seen in the lab was some dirty code written by a developer to let his joystick toggle through different values in the registers of the VGA. When he got the results he liked the best, he just jotted down the values off his secondary debug monitor and used these settings. I, for one, tend to use the calculator and reference material, but I thought this was the most whimsical technique I had ever seen.

No matter how you choose to get your values for programming the VGA card, one notion does remain clear. In order to get the results you desire for smooth game animation you, the programmer, must have access to the hardware in your code. Even direct access to modes like the BIOS-supported Mode 13h can be inadequate and have to be tweaked for the performance that a game requires.



Windows Video Abstractions

You have examined ways DOS programmers get high performance out of the VGA by accessing the video registers directly. But how is it then that the Windows 95 Game SDK can exceed DOS performance in frame-rates by factors of two and three? To answer this question you must understand how Windows 95 handles graphics in its native architecture, the Graphics Device Interface (GDI).



Windows 95 Graphic Device Interface

As you have seen working with DIBs in Chapter 2, a Windows programmer doesn't deal with low-level registers in the VGA card. Programmers who use the Microsoft suite of 32-bit C++ development tools already know that there are no software hooks provided by the compiler to examine and modify ports in the PC's hardware. The Windows 95 operating system handles all forms of graphical interfaces as abstractions. This means it doesn't matter what video card or printer you have; programming its usage through Windows is all the same. Making this interface properly to Windows 95 is pushed onto the manufacturer to provide the proper device drivers. When you buy a new video card you must install the device driver using the control panel. This abstraction, or virtualization of hardware, makes the Windows 95 user interface appear the same no matter what video hardware you may be using.

In Chapter 2, the concept of logical and physical spaces was introduced. You handled the movement of a sprite in logical space and translated this movement into the physical space of the DIB. The GDI behaves in a similar manner where the application is a logical space. The logical space of the application includes fonts, palettes, screen resolutions, pens, and drawing capabilities. The physical space in the GDI abstraction is the hardware or the devices. These devices, such as laser printers or super VGA cards, have unique ways information needs to be sent to them to print out a font or blit an image to the screen. This abstraction is shown in Figure 3-8.

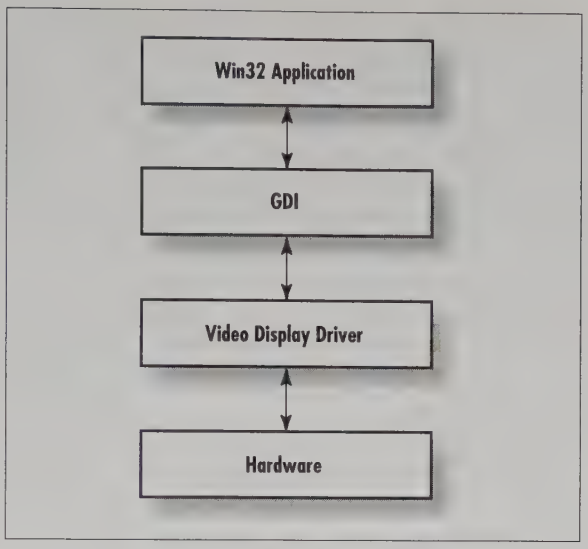


FIGURE 3-8
The Windows 95 GDI architecture

From Figure 3-8, you see that the GDI performs the actual communication between the Win32 application and the video display drivers. The display drivers handle the hardware-specific tasks that are required for a video card, printer, or plotter to perform a particular function. This is known as application buffering. A purist would also call this a software abstraction. But the idea to walk away with is that the GDI allows applications to behave in a device-independent manner. The application doesn't need to know the specifics of how the GDI communicates with the display driver. It just needs to understand it must hand off the information in a Windows 95-specific format like a DIB or a font, and the GDI will negotiate the rest with the lower layers. It is the manufacturers' responsibility to ensure their display drivers communicate properly with the GDI.

Using the logical space abstraction, a Win32 application like 13's DIB *PowerSuit* test can set the display's color palette. You used the API function call RealizePalette() with the appropriate parameters to set the palette you extracted from the DIB object. The palette is converted from a logical space object to a physical space object via a realization process, which maps the logical colors to the actual available colors on a VGA card. The GDI places the logical palette object into the physical palette object, which is a device-specific object containing the information the hardware actually requires to set the palette. If the hardware doesn't support certain functionality, the GDI will attempt to make its best guess and handle the functionality itself. An example of this is if the Bargain Mart VGA card can't perform hardware-level blitting, the GDI will emulate this in a manner the card can handle.

For application developers, like those writing e-mail packages, database tools, word processors, or any other information presentation package, Windows ensures a standard application interface across all platforms that Microsoft Windows 95 runs on. These developers are high-level developers using tools such as Visual Basic or Visual C++. Their concerns never exceed what is happening beyond the confines of the client application layer. The notion of frame-rates to these developers is pure nonsense. As the game developer, you view the client application as a wall you must hurdle. You care very little about opening documents; you want speed and you want to know how to bypass all the goodies a Windows 95 client application has to offer. You have one big problem—the Windows API (including the GDI layer) is in your way. A Win32 application must use the GDI to communicate with the graphical hardware. With the introduction of Visual C++ 2.0 and the removal of outp() and inp(), which are functions that communicate directly to hardware ports, Microsoft was pretty much saying they didn't want the developer messing with the lower layers. Besides, protected mode Windows applications can't directly access the hardware anyway due to the protected virtual address space given to each process.



Component Object Model

The notion of telling the developer to go away and not to touch anything in the lower hardware layer was the primary reason commercial developers abandoned the Microsoft Windows 3.x operating system as any form of suitable gaming platform. The Windows 95 Direct X Game SDK development team realized this was a significant problem they would have to deal with. The Game SDK would have to provide industry-level performance or better to get any attention from the commercial developer.

The obvious solution was that the GDI had to be bypassed in some way to access the hardware capabilities of VGA chip sets directly. A software architecture called the Component Object Model (COM) was already in use on various operating systems that handled this interpretability issue. The GDI is an integral component of the Windows 95 operating system, but the Game SDK team needed to provide device drivers from hardware vendors to integrate with other existing drivers in use by the GDI. The COM model provides a means to allow applications and systems to be built from executables supplied from different vendors.

The Microsoft Direct X Game SDK developers chose to provide an interface using their COM, since Microsoft's strategic direction is to provide an object-oriented API. This strategy allows the API to be accessible from different languages that understand how to interface to the COM. The Microsoft Direct X Game SDK DirectDraw interface is presented in the ddraw.dll.



What COM does for the programmer is provide an interface to these hidden modules of code, i.e., the device drivers. You're not wandering around the low levels of the hardware as you did with DOS, but now you're given a set of rules, or methods (functions) for how your code is expected to behave. Using these rules, you can instruct, through an interface, the low levels as to how you would like them to behave. In COM an interface is identified with an "I" naming convention. In the DirectDraw API you have the following object interfaces: IDirectDraw, IDirectDrawSurface, and IDirectDrawPalette.

One definition I have found amusing is that the interface is a contractual way for a component object to expose its services. This definition seems more like it came from the desktop of a lawyer than from a programmer. However you choose to word it, you are given a set of rules, which both the DirectDraw API developer and the OEM device driver programmer must adhere to. The display driver developer is going to present the proprietary features through the DirectDraw COM interface. The API developer is going to access these features through the set of rules defined in the interface.

So what is an interface in DirectDraw? An interface is just a related group of functions through which the application and the component objects (display drivers) communicate. For example, to set a palette in DOS you have to play with the DAC registers on the VGA card through ports. To set a DirectDraw palette, use the rules of the interface. The device dependencies of tweaking the proper ports on the VGA chip set are handled by the device driver. There is a transparency that is handled by the COM from the API to the device driver. The game programmer is only concerned with properly configuring the parameters at the function call level. The function call is the interface to the component object. The programmer is removed from the lower workings of the hardware and there is no cost in performance for this.

Interfaces to the COM are accessed through pointers, called interface pointers (also called object handles). The interface pointer can only access methods (functions) of the interface. Interface pointers cannot reference underlying data to the COM. This transparency encapsulates the behavior of the COM. The COM standard defines how the functions are laid out in memory. The interface methods exist in virtual function tables called *vtables*. Figure 3-9 shows a Windows 95 Game SDK application and its *vtable* pointer into the COM.

The Game SDK developers have further encapsulated these routines and you no longer make calls like this. However, what you do see in the above example is the COM interface method setDisplayMode() being accessed by the interface pointer *lpdd* via virtual table pointer *lpVtbl*. Early in the development of the Game SDK there was quite a bit of confusion as to why C language code accessed interfaces differently from C++ code, which didn't have to use the *lpVtbl* pointers to access an interface. C++ has its own virtual table mechanism and works rather well with the COM model. An example of C++ accessing a DirectDraw interface is provided in Listing 3-5.

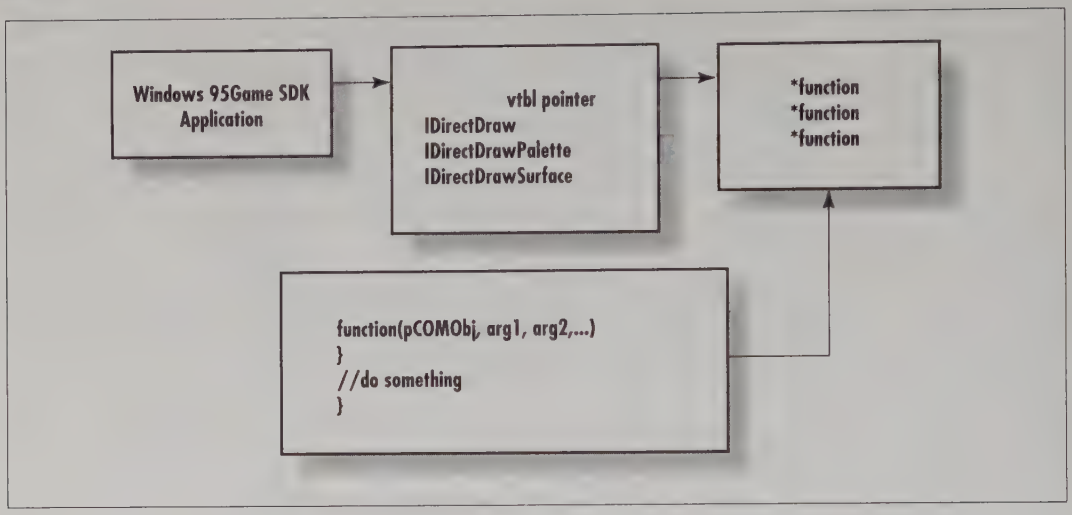


FIGURE 3-9 A Windows 95 Game SDK virtual table

LISTING 3-5 Accessing a DirectDraw interface using C++

```
LPDIRECTDRAW lpdd;
HRESULT ddrval;

/* set 640 x 480 8 bits per pixel */
ddrval = lpdd->setDisplayMode( 640, 480, 8);
```

You can see immediately in the C++ example that the virtual table pointer is implicit. Likewise, the COM interface setDisplayMode() object pointer *lpdd* is also now implicit. The COM interface for IDirectDraw is shown in Listing 3-6.

LISTING 3-6 IDirectDraw COM interface

```
DECLARE_INTERFACE_( IDirectDraw, IUnknown )
{
    /*** IUnknown methods ***/
    STDMETHOD(QueryInterface) (THIS_ REFIID riid, LPVOID FAR * ppvObj) PURE;
    STDMETHOD_(ULONG,AddRef) (THIS)  PURE;
    STDMETHOD_(ULONG,Release) (THIS) PURE;
    /*** IDirectDraw methods ***/
    STDMETHOD(Compact)(THIS) PURE;
    STDMETHOD(CreateClipper)
              (THIS_ DWORD, LPDIRECTDRAWCLIPPER FAR*, IUnknown FAR * ) PURE;
    STDMETHOD(CreatePalette)
              (THIS_ DWORD, LPPALETTEENTRY, LPDIRECTDRAWPALETTE FAR*,
                    IUnknown FAR * ) PURE;
    STDMETHOD(CreateSurface)
              (THIS_  LPDDSURFACEDESC, LPDIRECTDRAWSURFACE FAR *,
                    IUnknown FAR *) PURE;
```

```
    STDMETHOD(DuplicateSurface)
              ( THIS_ LPDIRECTDRAWSURFACE, LPDIRECTDRAWSURFACE FAR * ) PURE;
    STDMETHOD(EnumDisplayModes)
              ( THIS_ DWORD, LPDDSURFACEDESC, LPVOID,
                    LPDDENUMMODESCALLBACK ) PURE;
    STDMETHOD(EnumSurfaces)
              (THIS_ DWORD, LPDDSURFACEDESC, LPVOID,
                   LPDDENUMSURFACESCALLBACK ) PURE;
    STDMETHOD(FlipToGDISurface)(THIS) PURE;
    STDMETHOD(GetCaps)( THIS_ LPDDCAPS, LPDDCAPS) PURE;
    STDMETHOD(GetDisplayMode)( THIS_ LPDDSURFACEDESC) PURE;
    STDMETHOD(GetFourCCCodes)(THIS_  LPDWORD, LPDWORD ) PURE;
    STDMETHOD(GetGDISurface)(THIS_ LPDIRECTDRAWSURFACE FAR *) PURE;
    STDMETHOD(GetMonitorFrequency)(THIS_ LPDWORD) PURE;
    STDMETHOD(GetScanLine)(THIS_ LPDWORD) PURE;
    STDMETHOD(GetVerticalBlankStatus)(THIS_ LPBOOL ) PURE;
    STDMETHOD(Initialize)(THIS_ GUID FAR *) PURE;
    STDMETHOD(RestoreDisplayMode)(THIS) PURE;
    STDMETHOD(SetCooperativeLevel)(THIS_ HWND, DWORD) PURE;
    STDMETHOD(SetDisplayMode)(THIS_ DWORD, DWORD,DWORD) PURE;
    STDMETHOD(WaitForVerticalBlank)(THIS_ DWORD, HANDLE ) PURE;
};
```

The IUnknown is a special interface reserved by COM. All component objects are required to be derived from the IUnknown interface. IUnknown has three methods associated to it, QueryInterface(), AddRef(), and Release(). Using the IUnknown interface from Listing 3-6, the code looks like that provided in Listing 3-7.

LISTING 3-7 IUnknown COM interface

```
DECLARE_INTERFACE_( IDirectDraw, IUnknown )
{
  /*** IUnknown methods ***/
  STDMETHOD(QueryInterface) (THIS_ REFIID riid, LPVOID FAR * ppvObj) PURE;
  STDMETHOD_(ULONG,AddRef) (THIS)  PURE;
  STDMETHOD_(ULONG,Release) (THIS) PURE;
}
```

The QueryInterface() method is what is used at run-time to determine whether an interface is supported by the component object. When the Game SDK application wants to set the display mode, it calls the DirectDraw object's QueryInterface(), requesting a pointer to the method SetDisplayMode(). If the QueryInterface() fails, the object interface isn't supported and an error value is returned.

The calling parameter in QueryInterface(), REFIID, is a Globally Unique Identifier (GUID) to the interface called an Interface Identifier (IID). The IID is a 128-bit integer and is an absolute unique number (no other system object's identifier matches it) to identify this particular interface. This unique identifier

prevents access collision of COM objects. In other words, you can't accidentally connect to the wrong interface. The GUIDs are located in the binaries of the COM object. These identifiers are used for binding at run-time to ensure proper connections to the right object components.

The IUnknown interface method AddRef() counts the number of component objects currently using the interface. The method Release() decrements the count when a component object is through with the interface. You should never call these functions directly. Instead, these are explicitly called by the method QueryInterface() itself for the interface's own bookkeeping and accounting.

The long list of STDMETHODs presented in the IDirectDraw methods listing are the DirectDraw COM interface declarations. These are the hooks provided by the Game SDK development team to provide high-performance access to low-level components in supported video cards. When I first saw this architecture, I, for one, was a total disbeliever. First of all, there wasn't any way that a Windows 95 game could exceed a frame-rate of 15 frames per second. If Microsoft could come up with a way of providing 320 x 240 mode with 256 colors at a frame-rate of 30 frames-per-second, I would listen. Well, folks, when the Game SDK team demonstrated their goods, they had frame-rates in the 40–60 range. That wasn't the real kicker. What was more impressive was that the screen mode was 640 x 480 with 256 colors, linear video memory, and it supported page flipping. Other cards with better features were cranking the same demo at significantly higher frame-rates. My jaw hit the floor. A few months later, I saw a demo of *Wing Commander III*. Using the Game SDK and live video animation, it was clocking frame-rates in the 90s!



DirectDraw HAL

The DirectDraw Hardware Abstraction Layer (DDHAL) is the device-dependent code supplied by the VGA chip set manufacturer or its driver developer. The DDHAL is either located in the display driver itself or exists as a Dynamic Link Library (DLL), which communicates with the driver through a proprietary interface. The DirectDraw system component communicates with the DDHAL. The programmer does not interface with the DDHAL. All interfacing is carried out through the DirectDraw component. This architecture is shown in Figure 3-10.

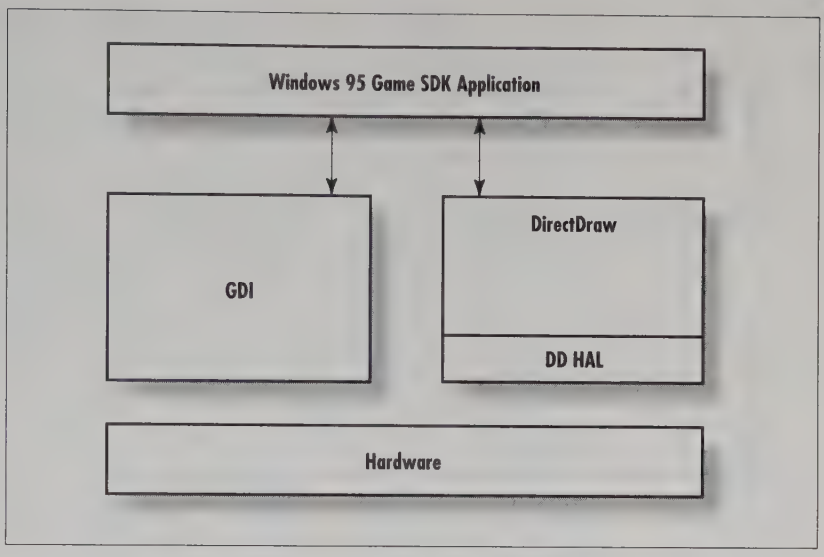


FIGURE 3-10
DirectDraw system component and DDHAL layer

DirectDraw

DirectDraw is the Microsoft Windows 95 Direct X Game SDK API that uses COM protocols to interface with the DirectDraw device driver layers supplied by chip set manufacturers. The DirectDraw API encapsulates all functionality required by the game designer, including virtual flat frame buffers for video cards with bank-switched memory, linear device-dependent memory called surfaces, unprecedented video frame-rates, and video hardware management. The Windows system DLL named DDRAW.DLL contains the DirectDraw functionality.

PC Architecture in Review

For DOS game programmers to get the performance they desire, they must code proprietary low-level device drivers that communicate directly with the ports and registers on the VGA card. A popular video mode among shareware developers is BIOS-supported Mode 13h, which has a resolution of 320 x 200, 256 colors, and a linear video memory buffer.

The commercial game developer is fond of Mode X, which has a resolution of 320 x 240, 256 colors, and a 4 x 64 KB plane memory scheme. This is a hybrid video mode and requires a lot of overhead to set up. Its greatest advantage is that it supports page flipping for a frame-rate in the range of 30–35 fps.

Graphics in the Windows 95 environment are accessed primarily through the Windows Application Program Interface (API). Device-independent graphics are supported through a subset to the API called the Graphics Device Interface (GDI). The Windows 95 GDI makes all calls to the display drivers and does not allow the programmer to make such low-level calls.

The Component Object Model (COM) provides an object-oriented interface that can be used by different languages and applications from different vendors. COM is the interface architecture selected by the Game SDK development team to provide access to low-level hardware components from higher-level languages. Programmers use interfaces to allow their software to interoperate with the supported display drivers. Interfaces are groups of functions supported by the component object. Game applications only interact with pointers to interfaces. COM interfaces used by the Game SDK for graphics support are IDirectDraw, IDirectDrawPalette, and IDirectDrawSurface.

DirectDraw is the system component to Windows 95 that presents the COM interfaces to the Windows 95 Game SDK application. DDRAW.DLL is the 32-bit component that provides all the functionality of DirectDraw. DirectDraw is the memory manager for video buffers and presents bank-switched VGA memory in the form of flat virtual memory.

DirectDraw HAL is the hardware dependent binaries that interface to DirectDraw. The DirectDraw HAL may be a device driver, a DLL, or both. The HAL implements the proprietary interface to the hardware and is supplied by the chip set manufacturer.



The DirectDraw API

Access to the DirectDraw COM interfaces are through the DirectDraw API. Each function in the API is a function located in the virtual table. You previously used the terminology method for a DirectDraw function. These two words are interchangeable and will be used as such in the remaining text. An object, in the sense of a DirectDraw object, is an object-oriented abstraction of the display device. Two subclasses exist in the DirectDraw object: the DirectDrawSurface, which is the video memory, and the DirectDrawPalette, which represents the video palettes. Figure 3-11 shows the DirectDraw architecture with the DirectDraw object with video surfaces and palettes.



Initializing a DirectDraw Object

DirectDraw objects completely encapsulate the functionality of the display card. This allows the Windows 95 Game SDK application to take over the

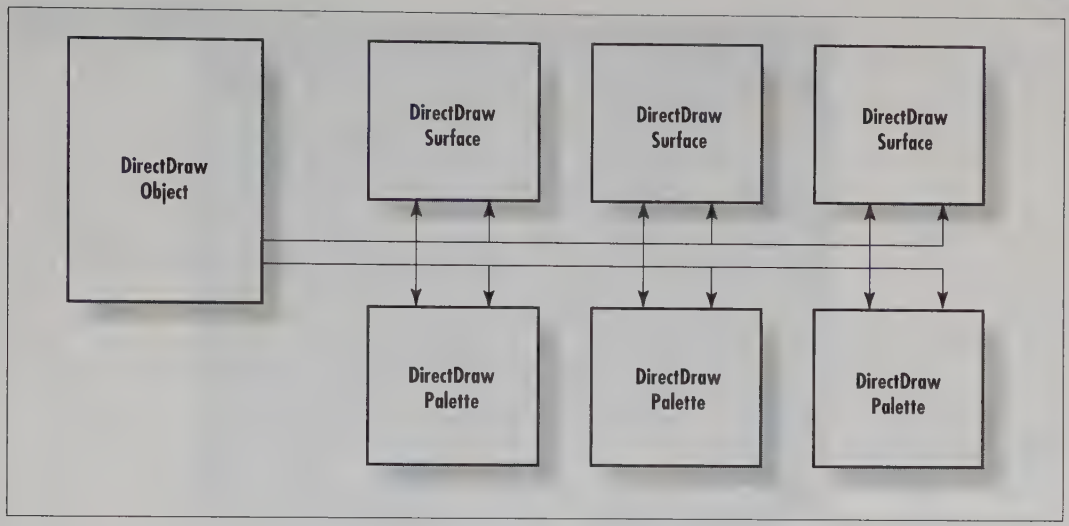


FIGURE 3-11 DirectDraw object architecture with surfaces and palettes

operation of the entire video card. Such an operation in DirectDraw terminology is called ExclusiveMode operation. In an ExclusiveMode operation, an application can manage the video card memory any way it sees fit. Direct access to video palettes is provided without having to use system memory palette buffers to write and read hardware ports. Resolution, bit depth, and page flipping can also be handled directly by the application. A DirectDraw object is object-oriented and an instance (a data object with values assigned) of the DirectDraw object must exist before any other DirectDraw API calls can be used.



Creating a DirectDraw Object

An instance of a DirectDraw object is created with the DirectDraw API call to DirectDrawCreate(). If the call to the function is successful, a DirectDraw object is created and a pointer to the object, LPDIRECTDRAW, is returned. Hooks are provided to use device drivers, other than those provided, for the primary display. These drivers are identified by a GUID (Globally Unique Identifier). The default driver for Windows 95 is the DirectDraw driver. Passing a NULL value to create a DirectDraw object initializes the driver for the Windows display device. Figure 3-12 shows the DirectDraw API function prototype for DirectDrawCreate(). Listing 3-8 provides the code for DirectDraw object instantiation for Windows 95 Game SDK applications. Note the first parameter is a NULL value for the display device GUID. The NULL value indicates that the display device is shared with the GDI.

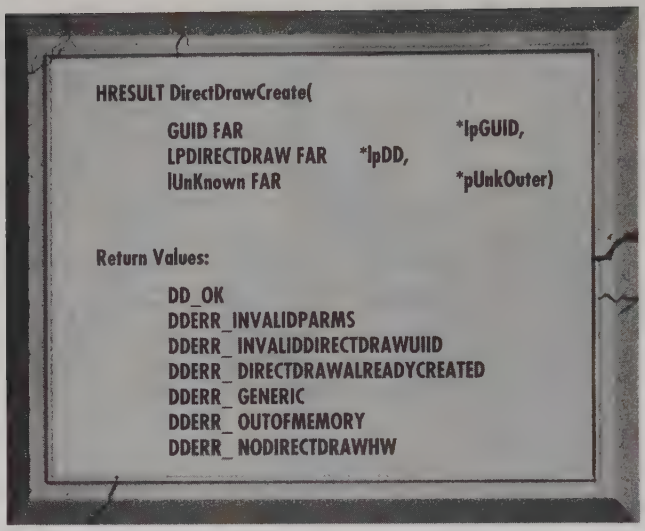


FIGURE 3-12
DirectDraw interface for creating a DirectDraw object

LISTING 3-8 DirectDraw object instantiation

```
LPDIRECTDRAW   lpDD;
HRESULT ddrval;
ddrval = DirectDrawCreate( NULL, &lpDD, NULL );
if( ddrval != DD_OK )
{
  return DDError("DirectDrawCreate Failed");
}
```

Standard error codes that might be returned by DirectDrawCreate are listed in Figure 3-12. The call seldom fails unless the system is out of memory or the wrong driver is present for the display device. The routine DDError() is a function provided in this book's library source, DDTOOLS.CPP. It is helpful to decipher error messages and display them in a Windows 95 message box call when the DirectDraw object instantiation fails. DDError() displays a message passed in as a calling argument after it shuts down the DirectDraw interface in the event of an error. The function is a generic routine used throughout in the DirectDraw examples in this book. If the DirectDrawCreate() call fails, there is no interface to shut down. However, if you get further along in your DirectDraw object instantiation and you run into a snag, you will have to clean up behind yourself as you back your way out. The utility DDError() is shown in Listing 3-9.

LISTING 3-9 DDTOOLS.CPP utility DDError()

```
BOOL DDError( char *errMsg )
{
  char errMsgBuf[80];

  DDShutdown();  // shut down DirectDraw
  wsprintf(errMsgBuf, errMsg);
  MessageBox(GetActiveWindow(), errMsgBuf, "DirectDraw Error", MB_OK);
  return FALSE;
}
```

Even though a DirectDraw object is instantiated successfully, there are still a few more operations that must be performed before the object can be put to use. You must set the cooperative levels for the COM interface and also set the video mode. You can consider the initialization of a DirectDraw object to be actually three steps. The first is the call to DirectDrawCreate(), which instantiates the DirectDraw object. Next, you must set the cooperative level. Finally, you set the video mode. When these three steps have been completed, the DirectDraw object is then initialized and ready for use.



Setting Cooperative Levels

In the Windows 95 environment, standard applications are friendly and share resources. Games are more like a toddler. They don't want to share anything; they want complete control. Using the Game SDK, complete ownership of resources is used when you create DirectDraw, DirectSound buffers, or DirectPlay objects. This tells Windows that you're no longer sharing and that you are dictating how things are going to behave until the game is paused (minimized) or terminated. For the DirectDraw object initialization, you are informing Windows 95 that you own the display device and not to do anything with it. This means as long as your game application is executing, no other Windows application can come along and change your display mode or palette. You remove the sharing functionality of Windows by setting the cooperative level of the game application. Figure 3-13 shows the DirectDraw interface for SetCooperativeLevel().

To set the cooperative level to an exclusive level, a flag or set of ORed flags must be passed to the DirectDraw method SetCooperativeLevel(). A typical application would use the set of ORed flags, DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN. The values of these flags are represented in Figure 3-14.

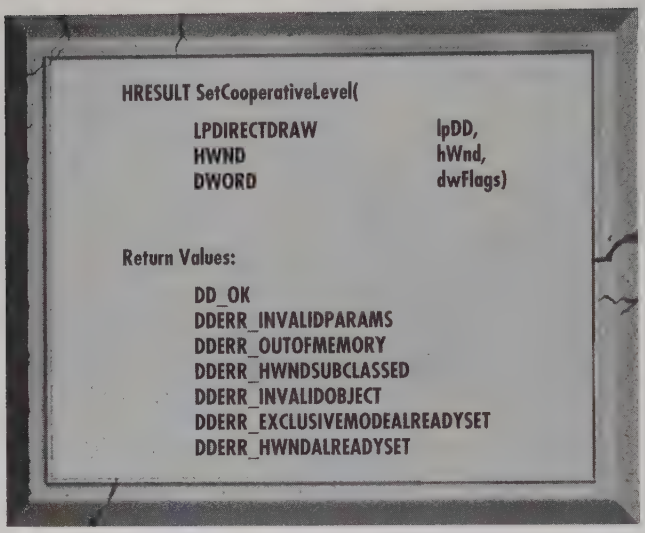


FIGURE 3-13
DirectDraw interface for setting ExclusiveMode

This particular combination gives your game application exclusive level and makes the application exclusive owner of the display memory, called a *primary surface*. (We'll get to that in a moment.) Exclusive mode means that other applications can't use DirectDrawSurfaces (DirectDraw video memory).

An example of using the DirectDraw method SetCooperativeLevel is shown in Listing 3-10. The HWND handle in the function call is the same HWND that Windows sends its messages to your game application.

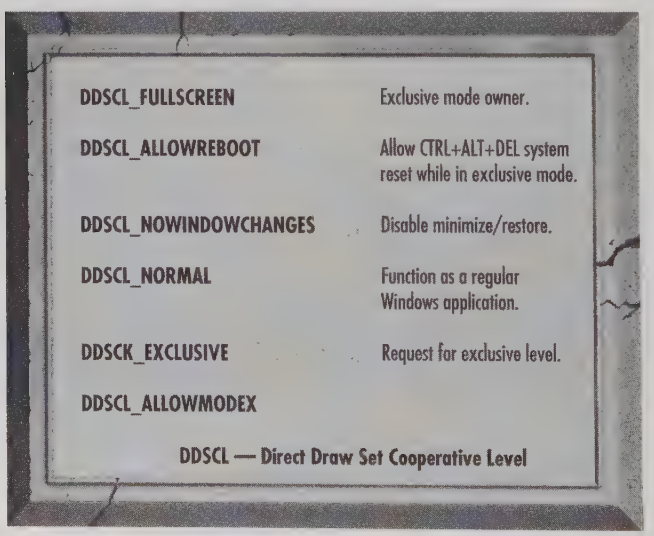


FIGURE 3-14
Flags for setting the game's cooperative level

LISTING 3-10 Setting the cooperative level for DirectDraw

```
ddrval = lpDD->SetCooperativeLevel( hwnd, DDSCL_EXCLUSIVE |
➥DDSCL_FULLSCREEN);
if( ddrval != DD_OK )
{
  return DDError("SetCooperativeLevel Failed");
}
```



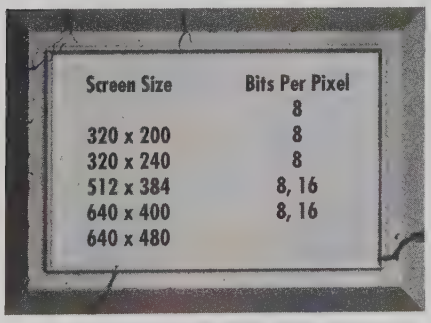
C++ and the DirectDraw COM Object PointerNote: Using C++, the object pointer to the interface is never passed to the interface method. For instance, in C, you would use the object pointer lpDD for SetCooperativeLevel(), lpDD->lpVtbl->SetCooperativeLevel(lpDD, hwnd, DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN);

The cooperative level must be set to exclusive before the display mode can be set. A mistake I commonly made with the first two applications I wrote for the Game SDK was that I would call CreateDirectDraw() and then SetDisplayMode(). The application would obviously fail because I was requesting exclusive level access to something the operating system had complete control over. This wouldn't cause a GPF error because the violation was in the interface access. Instead my application would just abruptly terminate. That's when I decided to just code a routine DDInit() that would handle the DirectDrawCreate(), SetCooperativeLevel(), and SetDisplayMode().



Setting Video Modes

The final step in initializing the DirectDraw object is to set the display mode. Figure 3-15 lists several display modes available in DirectDraw. Note that 320 x 200 and 320 x 240 screen modes are available with flipping functionality. The DirectDraw API allows complex video modes, such as 320 x 240, to be set in



Screen Size	Bits Per Pixel
	8
320 x 200	8
320 x 240	8
512 x 384	8, 16
640 x 400	8, 16
640 x 480	

FIGURE 3-15
Examples of screen modes available using DirectDraw

as little as three function calls. Since DirectDraw communicates directly to device drivers and hardware with accelerators, high-performance modes of 640 x 480 are also available.

A screen mode of 640 x 480, which is seldom used in DOS because of performance limitations, is now standard in the Windows 95 Direct X Game SDK. Figure 3-16 shows the method SetDisplayMode for the DirectDraw interface. A resolution of 640 x 480, with a bit depth of 8 bits per pixel, can be set for SetDisplayMode using the parameters shown in Listing 3-11.

LISTING 3-11 Setting a display mode using DirectDraw

```
ddrval = lpDD->SetDisplayMode( 640, 480, 8);
if( ddrval != DD_OK )
{
  return DDError("SetDisplayMode Failed");
}
```

In Figure 3-16, the first parameter passed to SetDisplayMode() is the width in pixels that is desired, followed by the height in pixels, and then the number of bits per pixel (bpp) of the display. Although it's common to set a bit depth of 8 bpp, some games are moving towards 16 bpp using the Game SDK API.

Setting the display mode is the final step in initializing a DirectDraw object. The complete listing for initializing the DirectDraw object is shown in Listing 3-12.

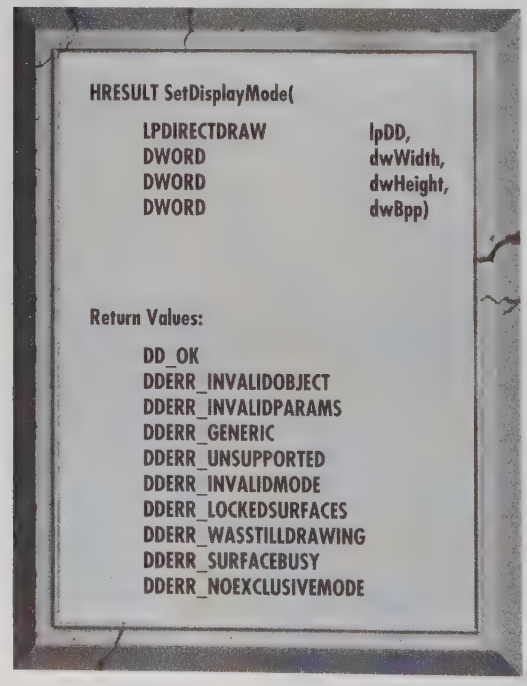


FIGURE 3-16
DirectDraw interface for setting display mode

LISTING 3-12 Initializing DirectDraw for 640 x 480 resolution with 8 bpp

```
BOOL DDInit(HWND hwnd)
{
  HRESULT ddrval;
  ddrval = DirectDrawCreate( NULL, &lpDD, NULL );
  if( ddrval != DD_OK )
  {
    return DDError("DirectDrawCreate Failed");
  }

  // Get exclusive mode
  ddrval = lpDD->SetCooperativeLevel( hwnd,
                  DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN);
  if( ddrval != DD_OK )
  {
    return DDError("SetCooperativeLevel Failed");
  }

  // Set the video mode to 640x480x8
  ddrval = lpDD->SetDisplayMode( 640, 480, 8);
  if( ddrval != DD_OK )
  {
    return DDError("SetDisplayMode Failed");
  }
  return TRUE;
}
```

Even though a DirectDraw object can be initialized in three easy steps, it's helpful to keep these in one canned routine. Using the function DDInit() in source file DDTOOLS.CPP and the Generic Windows 95 Game Application in GENX.CPP, game applications can be protoyped rather quickly. Using the function InitGameApp() from the source file GENX.CPP, you can initialize the DirectDraw object, as shown in Listing 3-13.

LISTING 3-13 Using DDTOOLS with the Generic Game Application from Chapter 1

```
BOOL  InitGameApp( HINSTANCE hThisInstance, int nCmdShow )
{
  HWND        hwnd;
  WNDCLASS    wndclass;
  char szAppName[] = "Generic Game SDK Window";

  wndclass.style         = CS_DBLCLKS;
  wndclass.lpfnWndProc   = WndFunc;              // window function
  wndclass.cbClsExtra    = 0;                    // no extra count of bytes
  wndclass.cbWndExtra    = 0;                    // no extra count of bytes
  wndclass.hInstance     = hThisInstance; // this instance
```

continued on next page

continued from previous page

```
  wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION);
  wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW);
  wndclass.hbrBackground = GetStockObject (WHITE_BRUSH);
  wndclass.lpszMenuName  = NULL;
  wndclass.lpszClassName = szAppName;

  RegisterClass (&wndclass);

  hwnd = CreateWindowEx (
     WS_EX_TOPMOST,          // window style
     szAppName,              // window class name
     "Generic Windows 95 Game App",// window caption
     WS_VISIBLE | WS_POPUP, // window style
     CW_USEDEFAULT,          // initial x position
     CW_USEDEFAULT,          // initial y position
     640,                    // initial x size
     480,                    // initial y size
     NULL,                   // parent window handle
     NULL,                   // window menu handle
     hThisInstance,          // program instance handle
     NULL);                  // creation parameters

  if (!hwnd)
    return FALSE;

  ShowWindow (hwnd, nCmdShow);
  UpdateWindow (hwnd);
  SetFocus(hwnd);
  ShowCursor(FALSE);

  // initialize the DirectDraw object
  if (DDInit(hwnd) == FALSE)
  {
    DDShutdown();
    MessageBox(hwnd, "DDInit Failed", "Error", MB_OK);
  }
  return TRUE;
}
```

The InitGameApp() for initializing your Generic Windows 95 Game Application required little modification for initializing DirectDraw. The code for DDInit() plugged right in. The DirectDraw object will initialize using this code, but you really can't do anything with it yet. You need to perform the same operations as you did with DOS video device initialization. You need to set up video memory buffers and palettes.



DirectDraw Surfaces

As has been alluded to many times thus far, a DirectDrawSurface is an abstraction of video memory. I feel the DirectDrawSurface is probably the greatest feature the DirectDraw development team has delivered to the game developer. The DirectDrawSurface is linear (flat) video memory. Remember when I discussed setting video modes in DOS, that 320 x 240 and 640 x 480 were arranged in planes of memory? Well, now the memory is flat, like DOS video Mode 13h.

There are a few limitations and requirements for using the DirectDrawSurface. First of all, those of you with the Bargain Mart video cards are going to have to sit in the back of the class. I have two 486-66 machines I use at home for development. One machine, a Dell, is (actually was) a state-of-the-art cruncher, with 64 MB of memory and an ATi Mach 64 video card with 2 MB of video RAM. My other system is my beater system: nothing matches, no name brands; it serves its purpose as my networked lab rat. The beater system has a Bargain Mart video card. So what has this got to do with DirectDrawSurfaces? My ATi has plenty of video memory and its drivers support the DirectDraw architecture. The beater system, with its caseless open air induction cooling system, has a video card with only 1 MB of memory. Suitable for running Windows 95, but responds with Direct what? When the DirectDraw tries to initialize the drivers, the Bargain Mart card must use system memory for multiple flipping surfaces and it must emulate many DirectDraw actions. Emulation, you all know, really means "slow at providing the desired results." Video memory to video memory transfers are fast and system memory to video memory transfers are slow.

The beater machine can still use DirectDraw because when the DirectDraw object is instantiated it implicitly creates a GDIPrimarySurface. Before the instantiation, the GDI is already using the display memory for the Windows 95 operating system. At the time of the instantiation, both the DirectDraw and the GDI will be pointing at the same surface of memory. The GDI will only know about the primary surface and have no knowledge of other surfaces. On Bargain Mart VGA cards the primary surface is typically created in video memory, while the back buffer resides in system memory.

DirectDraw operates under the assumption it is sharing resources with the GDI. The GDIPrimarySurface is the memory used by the GDI and is maintained by DirectDraw. GDI activities can go on in the background, but it is not taking effect on the primary game video memory, called the primary surface.

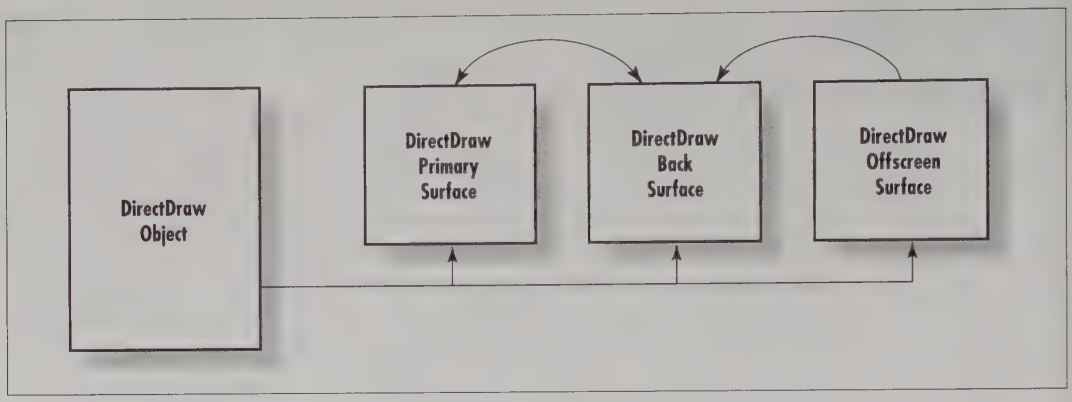


FIGURE 3-17 DirectDrawSurface architecture for video games

For video game applications, you will need to create at least three surfaces to perform video game animation. You will need a primary surface, a back surface, and an offscreen surface. This architecture is shown in Figure 3-17. The primary and back surfaces have attachment links for surface flipping. The offscreen surface can be used to update the back surface with sprites and redrawing backgrounds.

For console-style game surfaces, the size of the video display or larger may be initialized depending on your needs. For instance, you may set up your primary surfaces as 640 x 480 with an offscreen surface of 640 x 800 to hold your sprites and tiles.



Creating a Primary Surface

Like the DirectDraw object, the DirectDrawSurface is also object oriented. It needs to be instantiated just as you did with the DirectDraw object. There are many different combinations of DirectDrawSurface objects that can be created. Some are very complex for 3D animation and can have z-Buffers (depth-sorted buffers). The standard implementation of DirectDrawSurface objects consists of a primary surface (which is the linear frame of memory that is currently being displayed), a back surface (which is used for surface (or page) flipping), and an offscreen surface for holding tiles, sprites, and other objects.

To instantiate a primary surface DirectDrawSurface object, use the DirectDraw method CreateSurface(). Figure 3-18 represents the DirectDraw interface CreateSurface() and identifies some common return values. In order to successfully create a primary surface, the DirectDraw object must already be instantiated. The function call in source file DDTOOLS.CPP will create the DirectDraw object for you or you may integrate the three steps for instantiating the DirectDraw object on your own (discussed above in Creating a DirectDraw object).

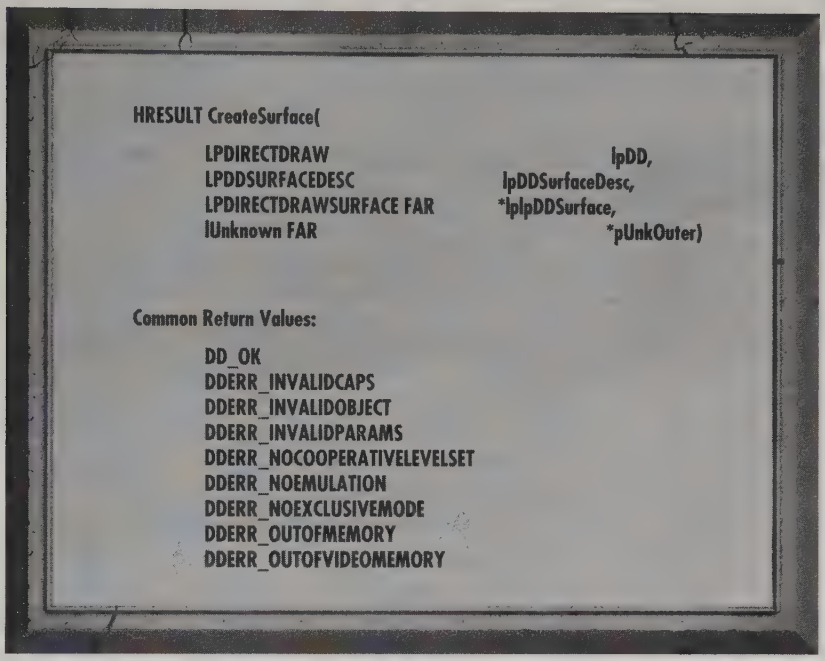


FIGURE 3-18
DirectDraw interface for creating a DirectDrawSurface object

Creating the primary surface requires the initialization of two DirectDraw data structures, DDSURFACEDESC (DirectDraw Surface Descriptor) and DDSCAPS (DirectDraw Surface Capabilities). The DDSCAPS fields must be filled in before a surface can be allocated. These fields describe the capabilities of the surface that is desired. About eighteen flags exist for initializing the DDSCAPS data structure but the most common flags are represented in Figure 3-19.

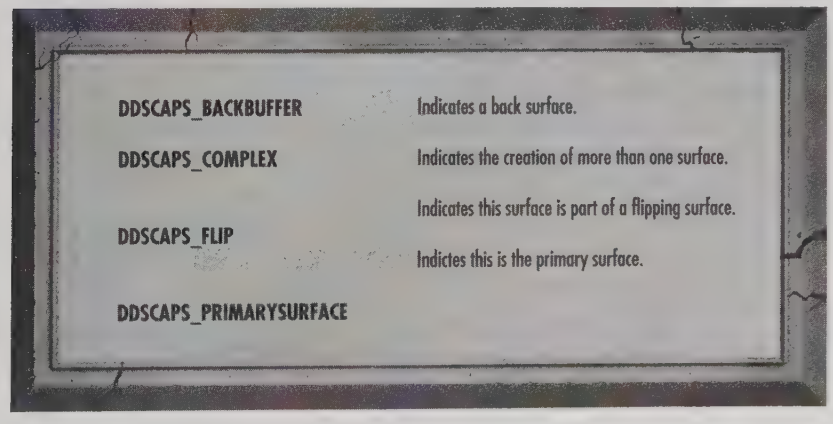


FIGURE 3-19
Common flags used for describing surface capabilities

To create a primary surface with an attached back surface for surface flipping, you need to specify a complex surface in the DDSCAPS data structure, as shown in Listing 3-14.

LISTING 3-14 Creating a complex DirectDrawSurface using DDSCAPS structure

```
DDSCAPS             ddscaps;

ddscaps.dwFlags = DDSCAPS_PRIMARYSURFACE | DDSCAPS_FLIP | DDSCAPS_COMPLEX;
```

These flags identify that you are creating a primary surface (DDSCAPS_PRIMARYSURFACE), with surface flipping capability (DDSCAPS_FLIP) and that it will have an attached back surface that you will use for the flips (DDSCAPS_COMPLEX). The flag DDSCAPS_COMPLEX doesn't identify the back surface; it represents that you will have a complex surface structure.

The DDSCAPS data structure is a field in the DDSURFACEDESC (DirectDraw Surface Descriptor) data structure. This contains the information you will be using when you call the DirectDraw method CreateSurface. Initializing DDSURFACEDESC for instantiating the Primary DirectDrawSurface object is shown in Listing 3-15.

LISTING 3-15 Creating a primary surface with an attached back surface

```
DDSURFACEDESC       ddsd;
HRESULT             ddrval;

// Create the primary surface with 1 back buffer
memset( &ddsd, 0, sizeof(ddsd) );
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |DDSCAPS_FLIP
➥|DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 1;
```

The DDSURFACEDESC fields *dwFlags* and *ddsCaps* are where you tell CreateSurface() the surface you are requesting. The field *dwFlags* tells DirectDraw what fields you are identifying in the DDSURFACEDESC data structure. Think of it as the checklist of functionality you want in your surface. Here you have checked the boxes DDSD_CAPS and DDSD_BUFFERCOUNT by ORing these flags in the *dwFlags* field. Next, fill in the desired functionality in the fields that correspond to the ones you identified in *dwFlags*. The field *ddsCaps.dwFlags* is where you request the primary surface, via the DDSCAPS data structure. The

last field you need to fill, based on your functionality questionnaire, is *dwBackBufferCount*. A DirectDrawSurface object will support *n* back surfaces, but you are only requesting one back surface. (As a homework assignment, I want you to go home and count them all.) Filling in these fields, you can make your call to CreateSurface(), using the DDTOOLS.CPP source presented in Listing 3-16.

LISTING 3-16 Calling CreateSurface() using *Spells of Fury* DDTOOLS library

```
DDSURFACEDESC       ddsd;
DDSCAPS             ddscaps;
HRESULT             ddrval;

// Create the primary surface with 1 back buffer
memset( &ddsd, 0, sizeof(ddsd) );
ddsd.dwSize = sizeof( ddsd );
ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |DDSCAPS_FLIP |DDSCAPS_COMPLEX;
ddsd.dwBackBufferCount = 1;
ddrval = lpDD->CreateSurface( &ddsd, &lpDDSPrimary, NULL );
if( ddrval != DD_OK )
{
  return DDError("Create lpDDSPrimary Failed");
}
```

In creating the primary surface, your options specified a COMPLEX surface. This means you have a primary surface with one or more attached surfaces. In your case, you requested one back surface in the DDSURFACEDESC field *dwBackBufferCount*. The DirectDraw method CreateSurface() handles the initialization of the primary surface and one back surface as you specified. You now need to use the DirectDrawSurface method GetAttachedSurface() to create a link between your primary and back surfaces. This link, or attachment, needs to be in place for surface flipping. Figure 3-20 defines the DirectDrawSurface interface for attaching surfaces, GetAttachedSurface().

Just as you did when you created the primary surface, you need to inform DirectDraw of the surface capabilities using the DirectDraw data structure DDSCAPS. Using the field *dwCaps*, you will request to attach your back surface using the DDSCAPS flag DDSCAPS_BACKBUFFER. Other flags are available for different styles of surfaces. See Figure 3-21 for usage.

Attaching the back surface requires only two lines of code. You need to first request the surface capabilities, DDSCAPS_BACKBUFFER, to attach a back surface. Then call the DirectDrawSurface method GetAttachedSurface(). Using the example from the source file DDTOOLS.CPP, this is shown in Listing 3-17.

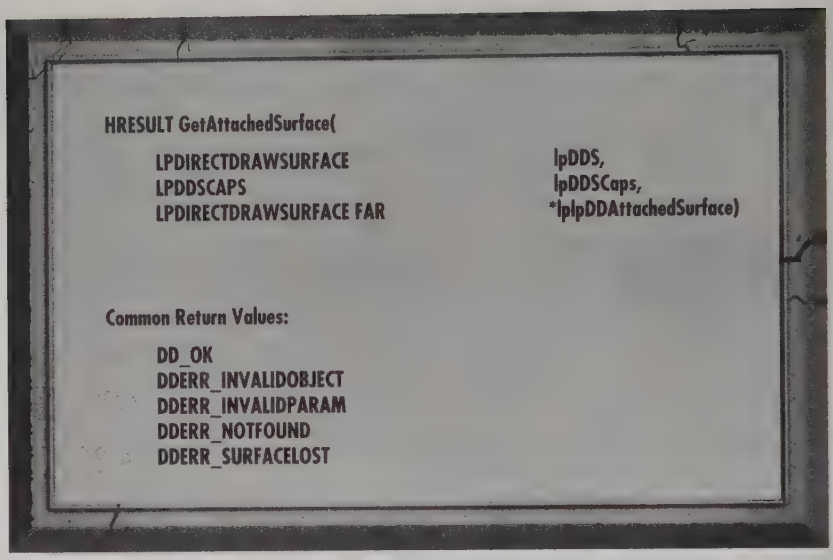


FIGURE 3-20
DirectDrawSurface interface for attaching surfaces

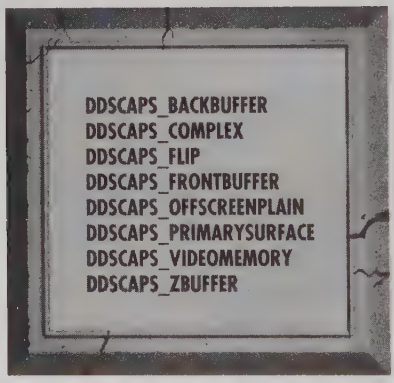


FIGURE 3-21
Common flags used for getting attached surfaces

LISTING 3-17 The DirectDraw method GetAttachedSurface()

```
ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
ddrval = lpDDSPrimary->GetAttachedSurface(&ddscaps, &lpDDSBack);
if( ddrval != DD_OK )
{
  return DDError("Attach lpDDSBack Failed");
}
```

The complete canned version of code used by DDTOOLS to instantiate the DirectDrawSurface objects is the function DDSInit(). This is the code used with the game examples in this book. The function DDSInit() is called after DDInit(). It initializes a primary surface and one attached back surface. The size of the surfaces are 640 x 480, 8 bits per pixel, which is the default resolution of the DirectDraw primary surface. The code for DDSInit() is shown in Listing 3-18.

LISTING 3-18 Initializing a DirectDrawSurface with *Spells of Fury's* DDSInit() routine

```
LPDIRECTDRAW   lpDD;
LPDIRECTDRAWSURFACE lpDDSPrimary;
LPDIRECTDRAWSURFACE lpDDSBack;

BOOL DDSInit()
{
  DDSURFACEDESC       ddsd;
  DDSCAPS             ddscaps;
  HRESULT             ddrval;

  // Create the primary surface with 1 back buffer
  memset( &ddsd, 0, sizeof(ddsd) );
  ddsd.dwSize = sizeof( ddsd );
  ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
  ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |DDSCAPS_FLIP
  ➥|DDSCAPS_COMPLEX;
  ddsd.dwBackBufferCount = 1;
  ddrval = lpDD->CreateSurface( &ddsd, &lpDDSPrimary, NULL );
  if( ddrval != DD_OK )
  {
     return DDError("Create lpDDSPrimary Failed");
  }

  // attach the Back surface
  ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
  ddrval = lpDDSPrimary->GetAttachedSurface(&ddscaps, &lpDDSBack);
  if( ddrval != DD_OK )
  {
    return DDError("Attach lpDDSBack Failed");
  }
  return TRUE;
}
```

Our tools now include a function for initializing a DirectDraw object, DDInit(), and a function for initializing primary and back flipping surfaces. You need to include in your tool bag a means to store images on surfaces. You could keep your images in DIB objects; however, the DIB is system memory, and transfers from system memory to video memory are slow. It would be nice to keep all of your images in video memory (a surface) as well.



Creating the Offscreen Surface

You're all dressed up but have nothing to blit. You need to create an offscreen surface for image storage. An offscreen surface in this context is just an area of linear memory that you use for retrieving images. A typical video card that supports DirectDraw functionality now ships with about 2 MB of video RAM. I know Microsoft is encouraging video card vendors to kick the standard up to

4 MB for 3D surfaces. But right now, for video game usage, 2 MB of memory is moderately generous for high-speed storage of sprites, backgrounds, and textures. If you're running a beater system like my caseless super air inductive-cooled game crunching beast, sporting a Bargain Mart VGA card with 1 MB of video memory, then this memory will be allocated from system memory in emulation mode.

For illustrative purposes let's create a 640 x 480, 8-bit-per-pixel depth surface (but an offscreen surface can be any size). You may want to create a wall texture surface of 64 x 128 for your own rendition of *Wolfenstein 3D* or even a large 640 x 800 offscreen surface to hold sprites, wall textures, and object tiles. Once you understand the mechanics of the offscreen surface you can pretty much use them in any way you want.

To create the offscreen surface, use the DirectDraw method CreateSurface(). This exercise is almost a repeat of creating a primary surface with a few differences. Remember how the data structure DDSURFACEDESC is the data waiter? Check off the items you want in the *dwFlags* field and fill in the specifics in those fields. DDSURFACEDESC then whisks off with the order to CreateSurface(). The fields you fill in for an offscreen surface will differ from the primary surface. For one, you don't want the surface to flip. (It won't be attached to any other surface.) Figure 3-22 shows how DirectDraw views these surfaces. Second, you want the offscreen to be arbitrary in size. This means you will have to check the boxes DDSD_HEIGHT and DDSD_WIDTH. You can fill in these fields on your own. You must also declare the surface as DDSCAPS_OFFSCREENPLAIN in the DDSCAPS field. Filling in these fields gives you the code in Listing 3-19.

LISTING 3-19 Describing an offscreen DirectDrawSurface using DDSURFACEDESC

```
DDSURFACEDESC       ddsd;
HRESULT             ddrval;

ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
ddsd.dwHeight = 480;
ddsd.dwWidth = 640;
```

That's basically all that needs to be done to initialize the DDSURFACEDESC field for an offscreen surface. Calling the DirectDraw method CreateSurface, the offscreen surface can be allocated. An example is shown in Listing 3-20.

LISTING 3-20 Creating the offscreen DirectDrawSurface

```
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSOffscreen, NULL);
if (ddrval != DD_OK)
        return DDError("Create lpDDSOffscreen Failed");
```

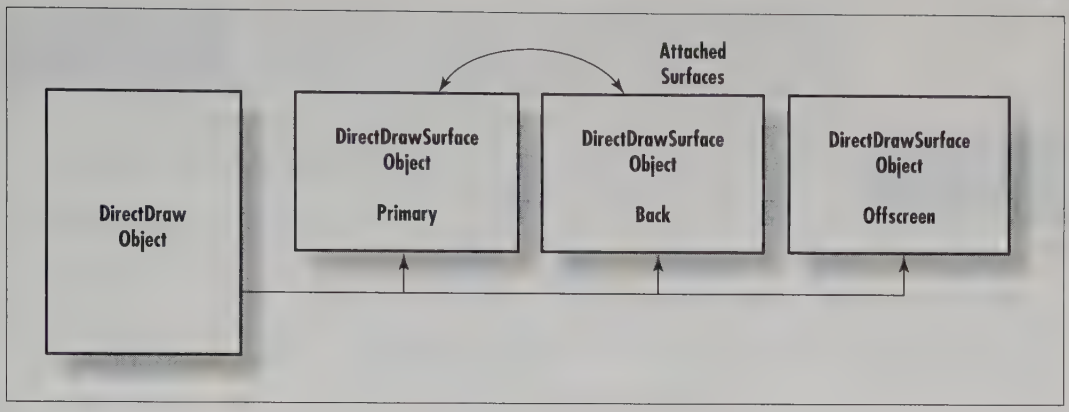


FIGURE 3-22 DirectDraw object and DirectDrawSurface objects

This was the brute force demonstration. I hard-coded the size of my offscreen surface. This doesn't provide too much flexibility in the actual game design. You want to be able to create multiple offscreens and with arbitrary dimensions. For your tool bag, this is the functionality you need to provide. The function GetNewSurface() in source file DDTOOLS.CPP handles these requirements. The function returns an LPDIRECTDRAWSURFACE pointer based on the dimensions passed into GetNewSurface(). The code is almost identical to what was presented above, except that it's canned and the height and width dimensions aren't hard-coded. This is shown in Listing 3-21.

LISTING 3-21 The *Spells of Fury* DDTOOLS routine to create an offscreen DirectDrawSurface

```
IDirectDrawSurface*  GetNewSurface( int width, int height )
{
  IDirectDrawSurface* lpDDSurface;
  DDSURFACEDESC       ddsd;
  HRESULT             ddrval;

  memset( &ddsd, 0, sizeof(ddsd) );
  ddsd.dwSize = sizeof( ddsd );
  ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
  ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
  ddsd.dwHeight = height;
  ddsd.dwWidth = width;
  ddrval = lpDD->CreateSurface(&ddsd, &lpDDSurface, NULL);
  if (ddrval != DD_OK)
  {
        return NULL;
  }

  return (IDirectDrawSurface*) lpDDSurface;
}
```

With this canned routine, you can use the CDib class from Chapter 2, read in a DIB object from a .BMP file, and dump the image to an offscreen surface. You can do this using the code in Listing 3-22.

LISTING 3-22 Creating an offscreen DirectDrawSurface using the CDib object from Chapter 2

```
pDib = new CDib;

if (!pDib->ReadFile("test.bmp"))
{
  MessageBox(hwnd,"BMP not found", "ERROR", MB_OK);
  return FALSE;
}

// create an offscreen surface from a DIB object
lpDDSOffscreen = GetNewSurface( pDib->GetCDibBiWidth(),
                                pDib->GetCDibBiHeight() );
// write the DIB object image bits to the surface
CopyDibToSurface( lpDDSOffscreen, pDib );
```

The CDib object pointer pDib reads in a .BMP file from disk. The CDib member function ReadFile fills out the information of the BITMAPINFO structure so you know the size of the DIB image. The height and width of the DIB image can be retrieved with the CDib member functions GetCDibBiHeight() and GetCDibBiWidth(). You can initialize the offscreen surface with these values:

```
lpDDSOffscreen = GetNewSurface( pDib->GetCDibBiWidth(),
                                pDib->GetCDibBiHeight() );
```

This will return a surface the size of the DIB object image. Now all you have to do is write the image bits into the surface using the DDTOOLS.CPP routine CopyDibToSurface(). When I first wrote this function I was thinking too hard and directly wrote image bits from system memory to the surface memory. But then one night, at 3 A.M., I was completely wired on my sixth can of Diet Coke and it dawned on me. The surface is video memory. If I can get the GDI to think it's got an HDC handle to it, I can use my CDib object member function DrawBits() to write into its context. So I downed another Diet Coke, flipped through the pages of the DirectDraw document and found the DirectDrawSurface method GetDC(). This method will lock the offscreen surface for me and create a compatible DC. To me, this meant I could use my CDib member DrawBits and blit my DIB object image into the offscreen surface. To do this I used the code in Listing 3-23.

LISTING 3-23 Copying the CDib object image bits to a DirectDrawSurface using a DC

```
BOOL CopyDibToSurface( LPDIRECTDRAWSURFACE lpDDS, CDib *pDibObj )
{
  HDC hdc;

  lpDDSOffscreen->Restore();

  if (lpDDSOffscreen->GetDC(&hdc) == DD_OK)
  {
      pDibObj->DrawBits(hdc, 0,0);
      lpDDSOffscreen->ReleaseDC(hdc);
  }

  return TRUE;
}
```

This particular function of DDTOOLS.CPP is used to walk both the DIB image bits and the surface using memcopy() in a loop and adjust pointers with new offsets at each loop pass. Now, thinking of all this in video memory terms, I just get a handle to the device context that is my offscreen surface and I merely blit the DIB into the surface. The CDib member DrawBits(), as you recall, is merely a canned function of the Windows API call StretchDIBits for the CDib object. I didn't perform any looping or row copying. For those of you who insist on doing things the hard way, the remnants of the original code can be pieced backed together using the DDTOOLS.CPP routine writeSurface().



Restoring Surfaces

In the tools code function CopyDibToSurface(), I called the DirectDrawSurface method Restore(). Even though you have privileged access to video memory in your games, you have no idea what the hardware is truly doing at the device driver level. Another scenario that is common with development groups working on one game is that one group may use the surface for composing before a flip, or set another exclusive mode and free your surface. In the event your surface memory is freed you may restore it with the call to Restore(). This method will reallocate surface memory and reattach it to the DirectDrawSurface object. Figure 3-23 shows the DirectDrawSurface interface for Restore.

This method is commonly used when you are composing a surface with a blit, or requesting a surface flip. If either of these operations return an HRESULT value of DDERR_SURFACELOST then the surfaces must be restored. A routine is provided in file DDTOOLS.CPP to restore lost surfaces, called restoreSurfaces(). This is demonstrated in Listing 3-24.

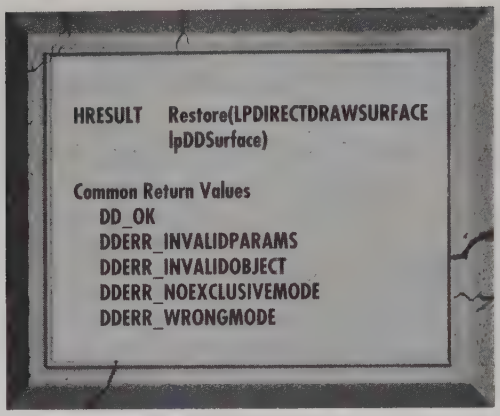


FIGURE 3-23
DirectDrawSurface interface for restoring freed surfaces

LISTING 3-24 Determining if a DirectDrawSurface is lost

```
HRESULT restoreSurfaces()
{
  HRESULT ddrval;

  ddrval = lpDDSPrimary->Restore();

  if (ddrval == DD_OK)
    ddrval = lpDDSBack->Restore();

  if (ddrval == DD_OK)
    ddrval = lpDDSOffscreen->Restore();
  return ddrval;
}
```

When I discuss blitting and flipping (in the next section), I will show precisely where to use this routine. Even in exclusive mode operation, you don't have exclusive control over your video memory and you should check to make sure you still have it before you write to it.



Blitting and Flipping Surfaces

I don't believe I'm any different from any other game programmer. All the books on my bookshelves represent my quest to design better blitters (whoa, that's

almost a Dr. Seuss sentence, the better blitter over butter battle), and an improved means of page flipping. This quest for performance has kept us glued to our PCs at home or in our labs for long hours. Hobbyists would have to shell out a good wad of cash to mail order the Super Quickie Draw tools from the back of a game programming trade journal. As a professional, you may recall being chained to your CPU by your manager to find a way to blow away the competition's astounding frame-rates.

Well, there is good news for the hobbyist. With the Windows 95 Game SDK you won't have to shovel out those precious dollars for Super Quickie Draw Professional. The power of page flipping and blitting is built in.

Let's take a look at how the next generation of computer games is going to be blitted across millions of video displays worldwide. The two DirectDrawSurface methods that make the DirectDraw API such a powerful game platform are BltFast() and Flip(). There is a broad range of video hardware that the DirectDraw COM must deal with. For my $39.95 Bargain Mart special VGA card I get a whopping 10 frames per second with surface flipping. On my lab system's configuration, a Dell 486/66, I get 60 fps on the ATi Mach 64 Graphics Xpression PCI bus card. I used the *Foxbear* sample on the Game SDK CD-ROM for these values. These figures are merely to demonstrate what I mean by a range of hardware. For the hobbyist and shareware developer, these values are acceptable tolerances. However, for those of you with managers, or worse yet, a marketing representative, guess where you're going to spend the remainder of your Game SDK development lives? Getting the Bargain Mart card to perform at 30 to 45 fps and finding out how to send the ATi and the S3 over the top with 100 fps.

You are currently set up for playing with surface flipping. The DDTOOLS.CPP routine DDSInit() provided you with everything you need to accomplish this. The criteria is that you have two surfaces that are attached with the following DDSCAPS flags set: DDSCAPS_FLIP, DDSCAPS_FRONTBUFFER, and DDSCAPS_BACKBUFFER. Initially your primary surface is set for the FRONTBUFFER, and the back surface is set for the BACKBUFFER. The idea of the surface flip is that the video memory that is set as the FRONTBUFFER becomes the BACKBUFFER and vice versa with each flip. For 3D and z-buffering you could have a ring of surfaces for composing and just cycle through these for updating the screen. However, for a simple picture of what is happening, Figure 3-24 shows how memory associations change with the call to the DirectDrawSurface method Flip().

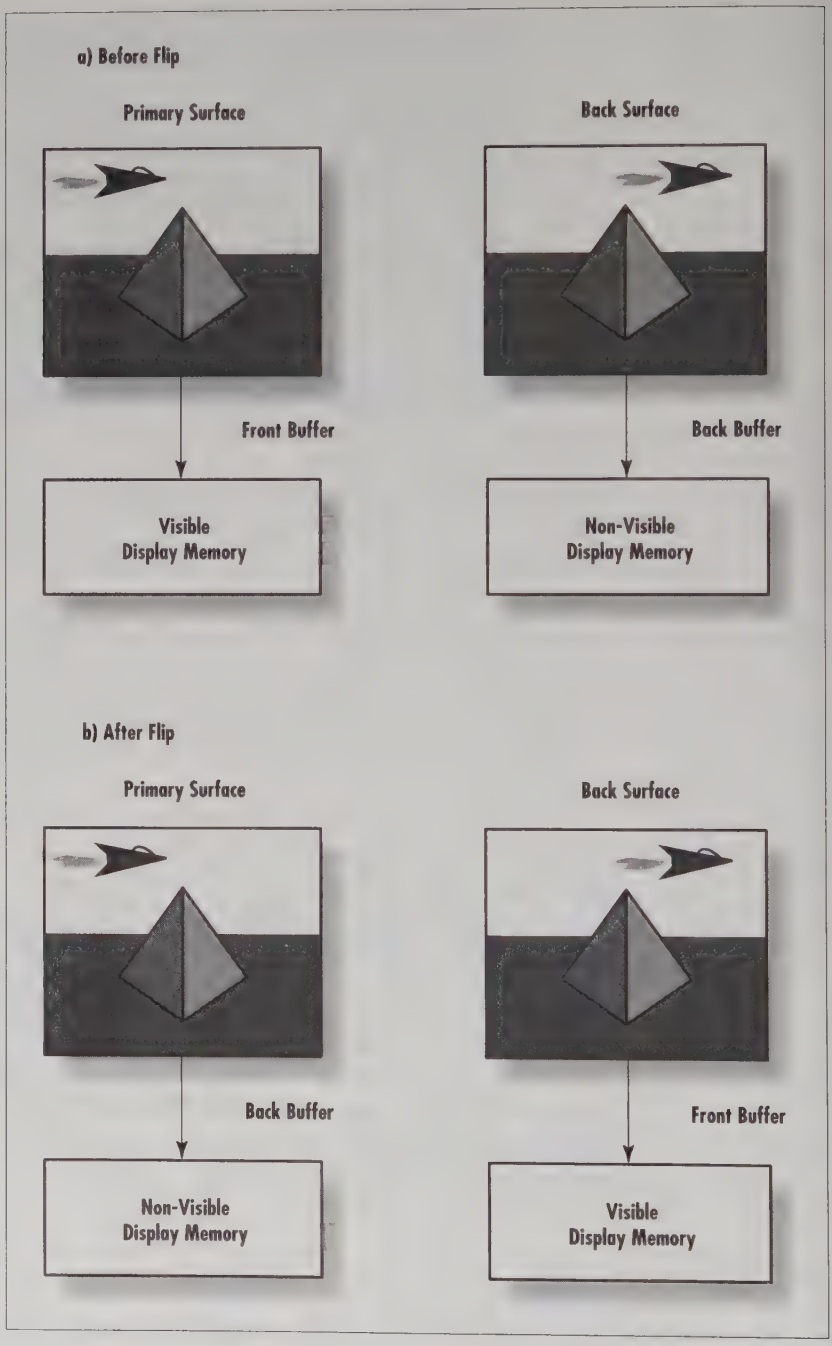


FIGURE 3-24
Surface flipping memory associations



The idea of page flipping is that you are cycling between two or more areas of memory. It's quite common in video game programming to use only two surfaces for arcade style composition. *Wolfenstein 3D* and *DOOM* use offscreen and primary video memory. To fully appreciate the levels of complexity Microsoft's Game SDK has kindly removed for the game developer, take a look at the *Wolfenstein 3D* source code, now available as shareware, to set up all the video card hardware level negotiations for a page flipping scheme. The DirectDrawSurface method Flip(), whose interface is shown in Figure 3-25, does this all for us.

The utility flipSurfaces() is provided in the source code DDTOOLS.CPP. This routine is callable only after both DDInit() and DDSInit() have been called successfully for DirectDraw object instantiation. The listing of flipSurfaces() is shown in Listing 3-25.

LISTING 3-25 The *Spells of Fury* DDTOOLS routine flipSurfaces()

```
void flipSurfaces()
{
  HRESULT ddrval = DD_OK;

  while( 1 )
  {
    ddrval = lpDDSPrimary->Flip(  NULL, 0 );
    if( ddrval == DD_OK )
      break;
    if( ddrval == DDERR_SURFACELOST )
    {
      ddrval = restoreSurfaces();
      if( ddrval != DD_OK )
         break;
    }
    if( ddrval != DDERR_WASSTILLDRAWING )
      break;
  }
}
```

The function is essentially an endless while() loop that isn't broken until the flip succeeds or fails. Why are you doing this? In your DOS environment, you the programmer are assured that you are the only register and memory manipulator of the VGA hardware. Things are a little bit more complicated in the Windows 95 operating system. Your own game is even capable of making simultaneous writes with different tasks working together to compose an image. One of these tasks may come in and lock the surface, manipulate the surface bits, unlock the surface, and leave. Another task, namely your flip, could be just right behind this task waiting to swap memory associations, but

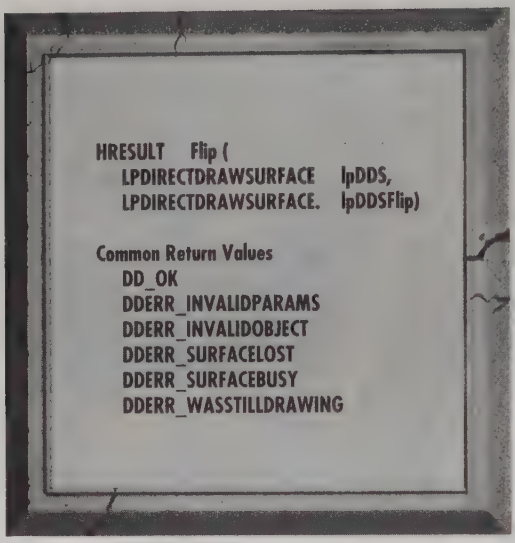


FIGURE 3-25
DirectDrawSurface interface for surface flipping

the task ahead of you hasn't unlocked the surface yet. The flip will keep attempting until it succeeds or decides it is incapable of performing the operation. The return value DDERR_WASSTILLDRAWING indicates the flip attempt was incomplete and terminated. When this is encountered, the flip is aborted. The return value DDERR_SURFACELOST indicates a flip can't be performed because the surface memory is freed. This error is handled by an attempt to restore the DirectDrawSurface object with a call to the DDTOOLS.CPP routine restoreSurfaces().

Implementing the flip routine is easy. If you want to play around and make a really wicked $3000 lava lamp, you could modify your message loop in source file GENX.CPP to look like the code provided in Listing 3-26.

LISTING 3-26 The Generic Game Application's message loop for use with DirectDraw

```
while(1)
{
  if(PeekMessage(&msg, NULL,0,0,PM_NOREMOVE))
  {
    if(!GetMessage (&msg, NULL, 0, 0))
      return msg.wParam;
    TranslateMessage (&msg);
    DispatchMessage (&msg);
  }
  else if (bIsActive)
  {
    // do critical game stuff
    flipSurfaces();
   }
}
```



This code will just blast out whatever random data happens to be in video memory to the display. It toggles between the primary and back surfaces. If you keep blinking your eyes and eat ice cream real fast in a dark room with this on, it would be kind of like a legalized psychedelic drug.

We've taken care of one complex aspect of video game programming, surface memory flipping, with a few simple calls. The art of composing screen images has also been significantly simplified. For those of you, like myself, who have had the pleasure of writing blitters for plane video memory schemes like Mode X, and 640 x 480 modes of resolution, you can now submit your code to the Smithsonian Institution. All surfaces in DirectDraw are linear. This means no more loading PCX or LBM files into system memory and reformatting the images for storage in video memory planes. The surface architecture is top-down, so the images appear to be stored in their natural formats. Blits between surfaces can be made using rectangular coordinates to identify the region you want to compose. Two significant DirectDrawSurface interfaces are available for composing images. These methods are Blt() and BltFast(). Other blitting and composing methods are available, but these are the two that are of the most interest to programmers. If your card supports hardware blitting, the difference between the two really isn't in the speed. The names are confusing. The method Blt() is actually your complex blitter for composing images that require surface fills, surface rotation, and alpha or z-buffer information. The DirectDrawSurface interface BltFast() is the blitter you'll be using about 85% of the time. This blitter is the one that performs source copy or transparent blits using the transparency color key. Transparency color keying was discussed in Chapter 2. The two DirectDrawSurface methods for blitting are shown in Figure 3-26.

As a coding example for DirectDrawSurface method Blt(), you can fill an entire surface with zeroes to erase it. A canned routine, EraseSurface(), in your tool bag, DDTOOLS.CPP, does just that. Remember your surfaces have garbage in them from whatever happened to be in memory when you initialized the surfaces. This function can easily be modified by adding a parameter for a color value and you'll have your own routine for flood filling a surface. Listing 3-27 shows the code for the *Spells of Fury* DDTOOLS routine EraseSurface().

LISTING 3-27 *Spells of Fury* DDTOOLS routine to erase a DirectDrawSurface

```
void EraseSurface(LPDIRECTDRAWSURFACE lpDDSurfPtr)
{
  DDBLTFX ddbltfx;
  ddbltfx.dwSize = sizeof(ddbltfx);
  ddbltfx.dwFillColor = 0;
  HRESULT ddrval;

  while (1)
  {
    ddrval = lpDDSurfPtr->Blt(NULL,NULL,NULL,DDBLT_COLORFILL,&ddbltfx);
```

continued on next page

continued from previous page

```
    if (ddrval == DD_OK)
      break;
    if (ddrval ==DDERR_SURFACELOST)
    {
      ddrval = restoreSurfaces();
      if (ddrval != DD_OK)
        return;
    }
    if (ddrval != DDERR_WASSTILLDRAWING )
      return;
  }
}
```

The EraseSurface() function looks very similar in operation to your flipSurfaces() routine. The function executes in an endless loop just as flipSurfaces() did. An interesting point to note is that in the call to Blt(), the source and destination RECT, as well as the source surface, are NULL values. DirectDrawSurface objects are object-oriented, so when you submit these fields with the value of NULL, the method assumes you are defaulting to the calling

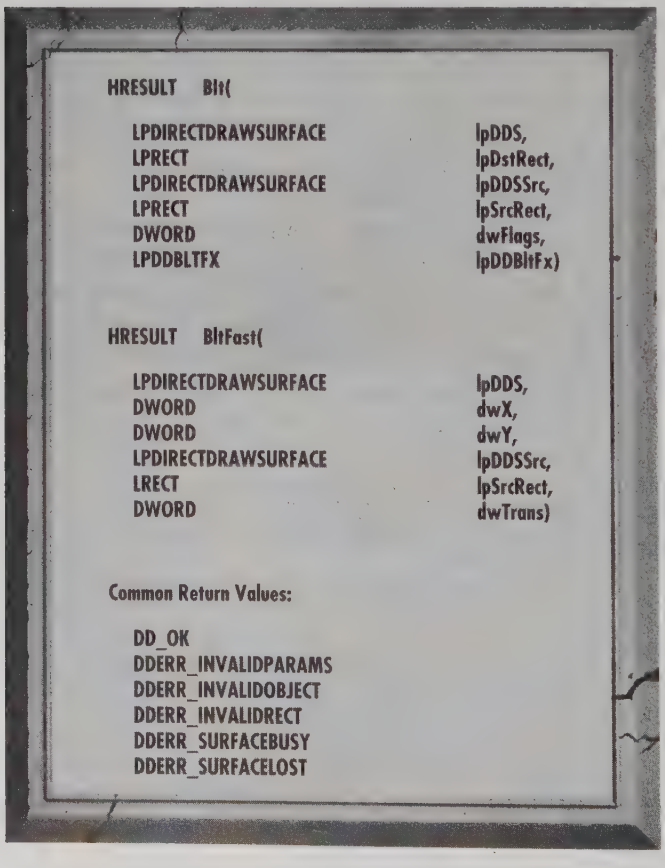


FIGURE 3-26

Two DirectDrawSurface interfaces for blitting



DirectDrawSurface object. The information for destination RECT will be filled in by object member data. The object that calls this interface will be erased (filled with color index 0).

In contrast to Blt(), the method BltFast() is used for blitting backgrounds and sprites. As you recall, BltFast() is the DirectDrawSurface interface that supports transparency color keying. This is the method you will use for patching backgrounds and blitting sprites between surfaces. The routine provided in DDTOOLS.CPP for transparency blitting is BlitSurface() as shown in Listing 3-28.

LISTING 3-28 *Spells of Fury* DDTOOLS routine to blit surfaces

```
BOOL BlitSurface( LPDIRECTDRAWSURFACE lpDDSSrc,
                          LPDIRECTDRAWSURFACE lpDDSDst,
                          DWORD dwX, DWORD dwY,
                          LPRECT lpSrcRect,
                          DWORD copyMode )
{
  HRESULT ddrval;

  while(1)
  {
       ddrval = lpDDSDst->BltFast( dwX, dwY, lpDDSSrc, lpSrcRect, copyMode);
       if ( ddrval == DD_OK)
           break;

       if (ddrval == DDERR_SURFACELOST)
         ddrval = restoreSurfaces();

       if (ddrval != DD_OK)
         return FALSE;

       if (ddrval != DDERR_WASSTILLDRAWING)
         return FALSE;
  }
  return TRUE;
}
```

These blitting loops are starting to develop a pattern, are they not? Figure 3-27 shows the functionality of BlitSurface(). In your offscreen DirectDrawSurface, you designate a RECT pointer to the region you want transferred. The source RECT is the source image's bounding rectangle. The destination rectangle for method BltFast() is the same size as the source rectangle (i.e., no scaling involved). For the destination of the image you merely provide the *x* and *y* (*dwX* and *dwY*, respectively) coordinates of where you want the image placed on the surface. If you are blitting a sprite, you will want to pass the flag DDBLTFAST_SRCCOLORKEY to indicate a transparent blit. If you are patching up a background's rectangular image, then use the flag DDBLTFAST_NOSRCCOLORKEY.

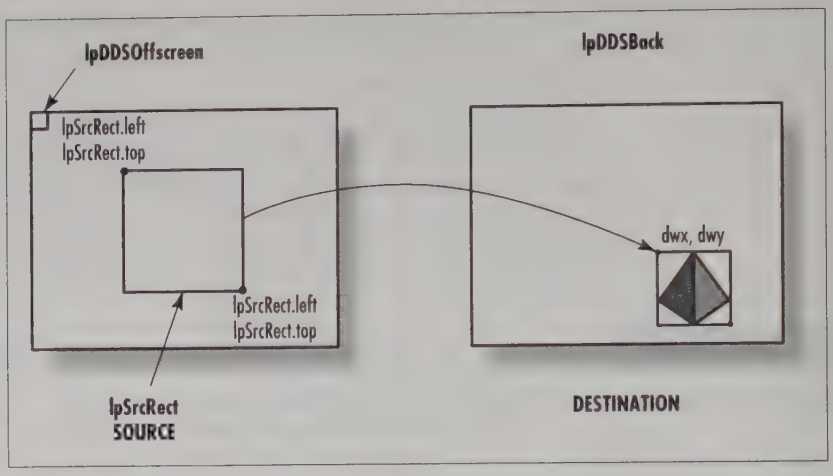


FIGURE 3-27
The mechanics of BlitSurface

The code looks pretty simple for blitting. We did momentarily jump ahead of ourselves. We haven't discussed palettes and source color keys. Setting the color key requires an understanding of another object in the DirectDraw architecture, the DirectDrawPalette.

DirectDraw Palettes

The DirectDrawPalette object provides interfaces for manipulating palettes attached to DirectDrawSurfaces. The palettes must be attached to the surfaces they are used by. Different surfaces may have different palettes. A DirectDrawPalette object is a 16-bit, 24-bit, or 8-bit 256-color indexed palette. The DirectDrawPalette is created by the DirectDraw object. This general architecture is represented in Figure 3-28.

Setting a Palette Object

If you were curious and played around with the DDTOOLS.CPP routine CopyDibToSurface(), you noticed the blit produced a very grainy looking image. The resulting image reminds me of the special effects used in the movie *Predator* to simulate the Predator's night vision. This effect occurred with CopyDibToSurface() because you didn't set the palette for the DirectDrawSurface. Sometimes you learn the coolest effects from your mistakes. No matter, you need to provide a DirectDrawPalette object to your surfaces so you can display your animation cells properly. You create a DirectDrawPalette object, just as you created the DirectDrawSurface objects from the DirectDraw object. The DirectDrawPalette object is also created from the DirectDraw object. The DirectDraw interface to create a DirectDrawPalette object is shown in Figure 3-29.

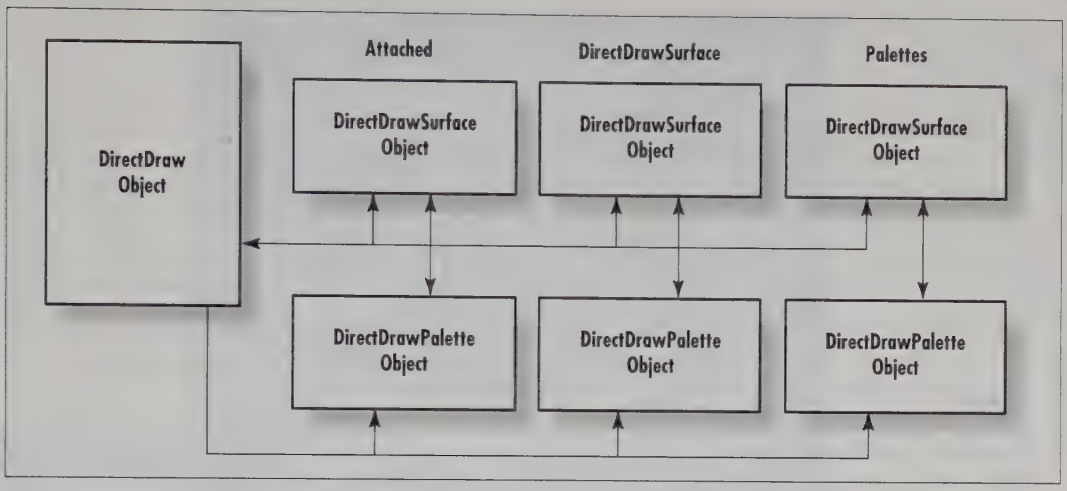


FIGURE 3-28 DirectDrawPalette objects and attached DirectDrawSurface objects

```
HRESULT CreatePalette (
        LPDIRECTDRAW                lpDD,
        DWORD                       dwFlags,
        LPDIRECTDRAWPALETTE FAR     *lplpDDPalette,
        LPALETTEENTRY               lpColorTable,
        IUnknown FAR                *pUnkOuter )

Common Return Values:

        DD_OK
        DDERR_INVALIDOBJECT
        DDERR_INVALIDPARAMS
        DDERR_NOCOOPERATIVELEVELSET
        DDERR_NOEXCLUSIVEMODE
```

FIGURE 3-29 DirectDraw interface for creating a DirectDrawPalette object

Once a DirectDrawPalette object is created, you need to attach the palette to a surface. The DirectDrawSurface interface to attach a palette to a DirectDrawSurface object is SetPalette(). A DirectDrawSurface object method, not a DirectDrawPalette object method, sets the palette. The method for setting a palette is shown in Figure 3-30.

Now you have created a palette object, but you don't have an indexed palette with the values you want. Remember the $3000 lava lamp you made when you initialized the primary and back surface and then flipped them? You had random garbage or leftover backgrounds toggling on the screen; basically,

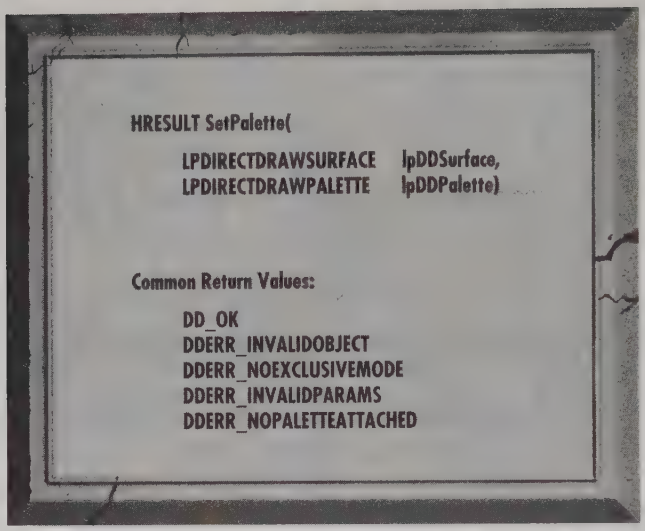


FIGURE 3-30
DirectDrawSurface interface for setting a palette to a surface

whatever happened to be in memory at the time you created the two surfaces. Well, the same is true for your palette object. You need to initialize the palette to the range of colors you desire. Just grabbing the system palette isn't going to work in video games unless you know of an artist that works specifically with the operating system palette. You need to retrieve the palette associated with your image. You're using DIB objects to store your images in system memory. The DIB object uses the Windows 95 RGBQUAD palette to store its entries. The DirectDrawPalette object has indexed palette entries, which is a different data structure from the RGBQUAD. The source file DDTOOLS.CPP provides a routine, SetRGBQuadPalette(), to set a palette that is stored as an RGBQUAD. This is shown in Listing 3-29.

LISTING 3-29 Setting the DirectDraw palette from a DIB RGBQUAD palette

```
void SetRGBQuadPalette(LPDIRECTDRAWPALETTE *lpPal, RGBQUAD *lpRGBQuad )
{
  int i;
  PALETTEENTRY pe[256];

  for (i = 0; i < 256; i++)
  {
    pe[i].peRed = lpRGBQuad[i].rgbRed;
    pe[i].peGreen = lpRGBQuad[i].rgbGreen;
    pe[i].peBlue = lpRGBQuad[i].rgbBlue;
  }
  lpDD->CreatePalette(DDPCAPS_8BIT, pe, lpPal, NULL);
}
```

Both the DirectDrawPalette object and the RGBQUAD store values as indexed palette entries. You merely need to loop through the 256 entries of the RGBQUAD table and write it to its respective location in the DirectDrawPalette object-indexed palette entry. The routine SetRGBQuadPalette() then uses the palette entry's array and calls the DirectDraw interface CreatePalette. For the DirectDrawPalette object instantiation, you use the flag DDPCAPS_8BIT to indicate there are 256 colors in the palette with 8-bit indexed entries. The palette entry's array is passed in along with a pointer to the DirectDrawPalette object you want to create. If the call to CreatePalette is successful, you will have an instantiated object ready to attach to your surfaces. The attachment can occur with a call to the DirectDrawSurface method SetPalette(). Some examples would look like the code shown in Listing 3-30.

LISTING 3-30 Setting the DirectDrawPalette from the CDib object

```
SetRGBQuadPalette(&lpDDPal,  pDib->GetLPRPGPtr());
lpDDSPrimary->SetPalette(lpDDPal);
```

These are calls you would most likely make in the function InitGameApp(), from source file GENX.CPP. If you are using any other surfaces, such as for texture storage or sprites, you will want to associate the palettes to these surfaces as well. You could have different DIBs stored in different surfaces with each surface having its own attached palette from the DIB's RGBQUAD.



Creating a ColorKey

Well I did leave you hanging about color keys. It's just that you needed to have a look at DirectDrawPalette objects before examining this aspect of blitting. The color key is identical to the transparency values discussed in Chapter 2. The color key is set in DirectDraw by the DirectDrawSurface method SetColorKey(). This interface is represented in Figure 3-31.

Setting the DirectDraw ColorKey data structure can be as easy as making a call to the DDTOOLS.CPP routine SetTransparentColor(). The code for SetTransparentColor() is shown in Listing 3-31.

LISTING 3-31 *Spells of Fury* DDTOOLS routine to set a DirectDraw ColorKey

```
BOOL SetTransparentColor( LPDIRECTDRAWSURFACE lpDDS, DWORD transColor )
{
  DDCOLORKEY ddColorKey;
  HRESULT ddrval;

  ddColorKey.dwColorSpaceLowValue = transColor;
  ddColorKey.dwColorSpaceHighValue = transColor;
```

continued on next page

continued from previous page

```
  ddrval = lpDDS->SetColorKey(DDCKEY_SRCBLT, &ddColorKey);
  if (ddrval == DD_OK)
    return TRUE;
  return FALSE;
}
```

This function accepts a surface and a color transparency value as input. The color transparency value is the byte you retrieved from a DIB image or hard-coded to black (value is 0). These values are set in the DirectDraw DDCOLORKEY data structure fields *dwColorSpaceLowValue* and *dwColorSpaceHighValue*. Since you are using 8-bit bytes to describe your transparency color value, you set both fields to the same value. After the fields are set, the DirectDrawSurface method SetColorKey() is called.

I typically grab the transparency color value directly from the DIB with the sprites on it. An example of this is similar to what you did in Chapter 2 with the DIB object member function GetPixelByte(). In fact I use GetPixelByte to retrieve my transparency color. An example of this is shown in Listing 3-32.

LISTING 3-32 Setting a DirectDraw ColorKey value from a CDib object pixel

```
SetTransparentColor(lpDDSOffscreen, (DWORD)pDib->GetPixelByte(0,20));
```

All the pieces are in place to start playing with some high-performance graphics. You have tools to initialize a DirectDraw object. With the DirectDraw

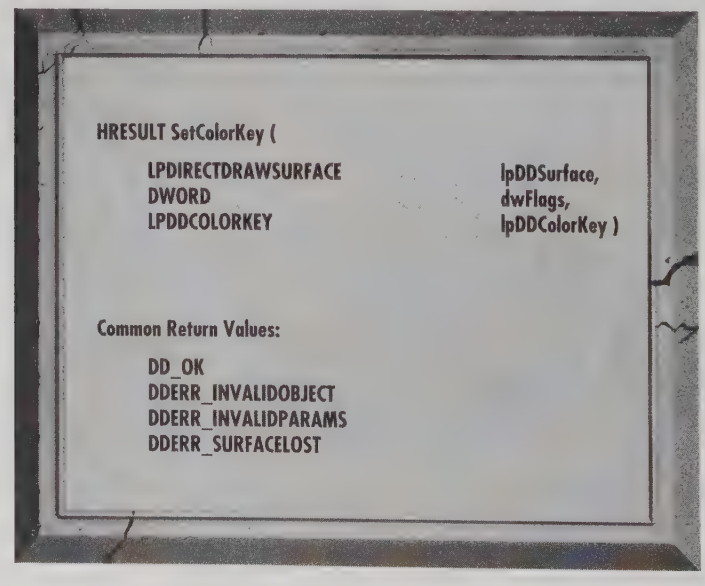


FIGURE 3-31
DirectDrawSurface interface for setting a ColorKey

object you can initialize and instantiate DirectDrawSurface and DirectDrawPalette objects. If you wanted to get up and running quickly with a dirty prototype you could use the code in Listing 3-33 for InitGameApp().

LISTING 3-33 Getting up and running quickly with *Spells of Fury* DDTOOLS and CDib Object

```
 pDib = new CDib;
 if (pDib->ReadFile("MyDIB.bmp") == FALSE)
 {
   MessageBox(hwnd, "ReadDib Failed", "ERROR", MB_OK);
   return FALSE;
}

 if (DDInit(hwnd) == FALSE)
{
  DDShutdown();
  MessageBox(hwnd, "DDInit Failed", "Error", MB_OK);
}
DDSInit();
eraseBackSurface();

//. create an offscreen surface from a DIB object
lpDDSOffscreen = GetNewSurface( pDib->GetCDibBiWidth(), pDib-
➥>GetCDibBiHeight() );
EraseSurface( lpDDSOffscreen );

// retrieve the RGBQUAD entries from the DIB
SetRGBQuadPalette( &lpDDPal, pDib->GetLPRPGPtr());
lpDDSPrimary->SetPalette(lpDDPal);
lpDDSBack->SetPalette(lpDDPal);
lpDDSOffscreen->SetPalette(lpDDPal);

// write the DIB object image bits to the surface
CopyDibToSurface( lpDDSOffscreen, pDib );

SetTransparentColor(lpDDSOffscreen, (DWORD)pDib->GetPixelByte(0,20));
```

You essentially have the basic components for putting together animation sequences for your games. With SetTransparentColor(), you can retrieve the transparency value from the DIB and set the value in DirectDraw for blitting your sprites. CopyDibToSurface() allows you to copy the art to a surface that you can display the images from. SetRGBQuad() allows you to set a 256-color indexed palette to a DirectDrawSurface object. You also have the necessary tools to initialize DirectDraw objects. Now roll up your shirt sleeves and put your tools to work.



Super PowerSuit Gaming on Dynamic Terrain

The limitations of Win32 GDI-based applications is that they don't have the performance required to update the entire game screen. In Chapter 2, 13 provided you with a coding example of using DIBs to animate virtual reality PowerSuits moving around Agamemnon's virtual world. If you recall, 13 divided his screen up into logical coordinates, 20 cells wide by 15 cells in height. These cells were each 32 x 32 pixels in size. 13's single PowerSuit was confined to this 20 x 15 cell world. Using the hardware assistance from DirectDraw, you can increase the PowerSuit range and create a virtual terrain of 60 x 60 cells. You will use tiles to scroll around and follow the PowerSuits as they explore the terrain map.



Programming with Tiles

Super PowerSuit allows two players to enter the virtual realm and, if they wish, engage each other in combat. The Super PowerSuit display is 640 x 480, but you will use a viewport of 480 x 480 pixels to view the terrain. The right side of the screen will provide status information on PowerSuits you are attacking. Figure 3-32 shows the Super PowerSuit's enhanced display.

Just as 13 did with the DIB PowerSuit game screen, you are going to divide your 480 x 480 game viewport into divisible 32 x 32 unit cells. This equates to a viewport of 15 x 15 cells. These cells will be your viewport into your 60 x 60 cell terrain map. The viewport isn't static like the DIB PowerSuit predecessor. You are going to slide the viewport, your looking glass into the virtual world, as you move across the terrain map. The terrain map is declared in source file globals.cpp like this:

```
SPOT    terrain[TERRAIN_HEIGHT][TERRAIN_WIDTH];
```

The terrain map is a two-dimensional array consisting of a data structure called a SPOT. A SPOT is a location on the map that is located with *x* and *y* coordinates. A SPOT has two distinct attributes. One, it identifies the tile, called a cell, occupying that location, and second, it is a pointer to a PowerSuit, if one is standing in this cell. The definition of a SPOT is shown in Listing 3-34.

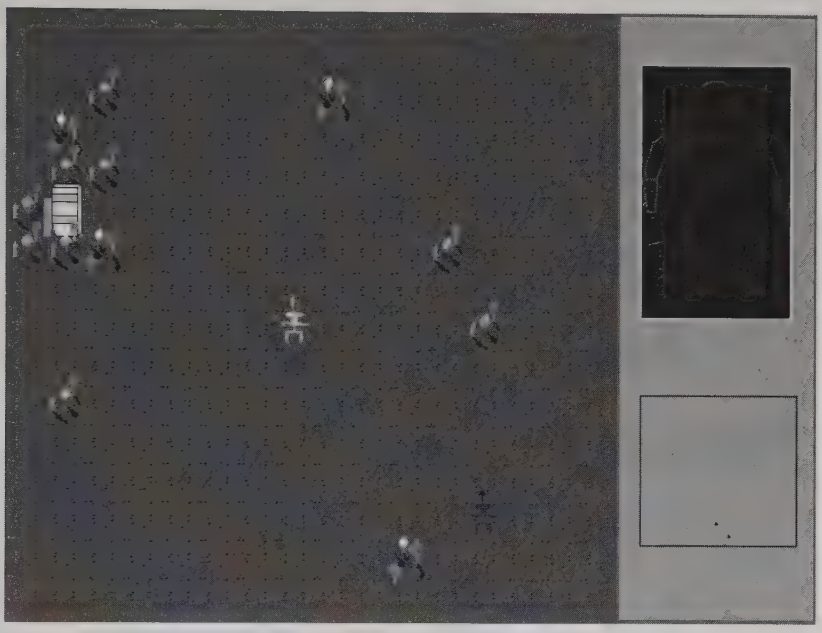


FIGURE 3-32
Display screen for Super PowerSuit

LISTING 3-34 Defining a location on a terrain map

```
#define TERRAIN_WIDTH   60
#define TERRAIN_HEIGHT  60

typedef struct {
  int        cell;         // The graphic cell to use
  PWRSUIT    *suitPtr;     // Pointer to powersuit that is in SPOT
} SPOT;
```

The SPOT data structure identifies the tile in each cell of the terrain array with a data structure called a CELL. A CELL is essentially a struct that identifies the attributes of a tile. It is declared in file 3-4PWR.H and is shown in Listing 3-35.

LISTING 3-35 Defining a CELL in the terrain map

```
typedef struct {
  int posx;
  int posy;
  int height;
  int width;
  int frames;
  int visibility;
} CELL;
```

Characteristics of a CELL are *x* and *y* coordinates, *posx* and *posy*. These represent the upper left corner of the tile in an offscreen surface. A tile is a rectangular object, with a width and height. Other attributes of the CELL are frame sequence number and visibility. In Super PowerSuit, an object may not be visible in the line of sight of a PowerSuit. So, from its frame of reference, the tile may be invisible. These fields introduce some complexities, so let's focus on the terrain generation and display using CELLs. What you need to know about CELLs is that they are tile attributes stored in the terrain map. When you slide your viewport around the terrain map, you will examine the CELLs to determine which tiles to display.

The actual tiles are initialized in source file GLOBALS.CPP as an array of struct CELLs. This declaration is shown in Listing 3-36.

LISTING 3-36 Tile definitions used in *PowerSuit*

```
CELL   cells[] = {
  0,0,47,31,1,0,          // grass                tile 0   cells[0]
  31,0,47,31,1, 1,        // light Trees          tile 1   cells[1]
  62,0,47,31,1,2,         // Heavy Trees          tile 2   cells[2]
  93,0,47,31,1,3,         // Building             tile 3   cells[3]
  0, 47, 47,31,1,0,       // Paved Road (E-W)     tile 4   cells[4]
  31,47, 47,31,1,0,       // Paved Road (N-S)     tile 5   cells[5]
  62,47,47,31,1,0,        // Intersection         tile 6   cells[6]
   0,94,57,31,1,3,       // Big Building part #1  tile 7   cells[7]
   31,94,57,31,1,3,      // Big Building part #2  tile 8   cells[8]
};
```

Each row of the CELL array cells[] consists of five entries. Each entry is a value for the corresponding five fields in the struct CELL. For cells[0] you have the values 0,0,47,31,1,0. These are *cells[0].posx = 0*, *cells[0].posy = 0*, *cells[0].height = 47*, *cells[0].width = 47*, *cells[0].frame = 1*, and *cells[0].visibility = 0*. The first four attributes describe the physical bounding rectangle of the tile in the offscreen surface. This so happens to be a tile, representing grass for the game field. The upper left corner of the tile is located in the offscreen surface at coordinates (0,0), which is the upper left corner of the surface itself. The width of the tile is 47 pixels and the height of the tile is 47 pixels. When you blit the tile you will use these attributes to compose a source RECT of coordinates (0,0,47,47). The destination RECT is arbitrary to wherever you are going to draw the tile based on the terrain map. The storage of the tiles is shown in Figure 3-33. Super PowerSuit has nine different styles of tiles. These include a tile for grass, a tile for grass with trees, a tile for woods, a tile for roads, and so on. This array can be expanded to include as many complex tiles as you desire.

The terrain map is a two-dimensional array of structure SPOT. As was discussed previously, one of the fields of struct spot is a CELL ID. When you build your terrain map it will look like the array shown in Listing 3-37.

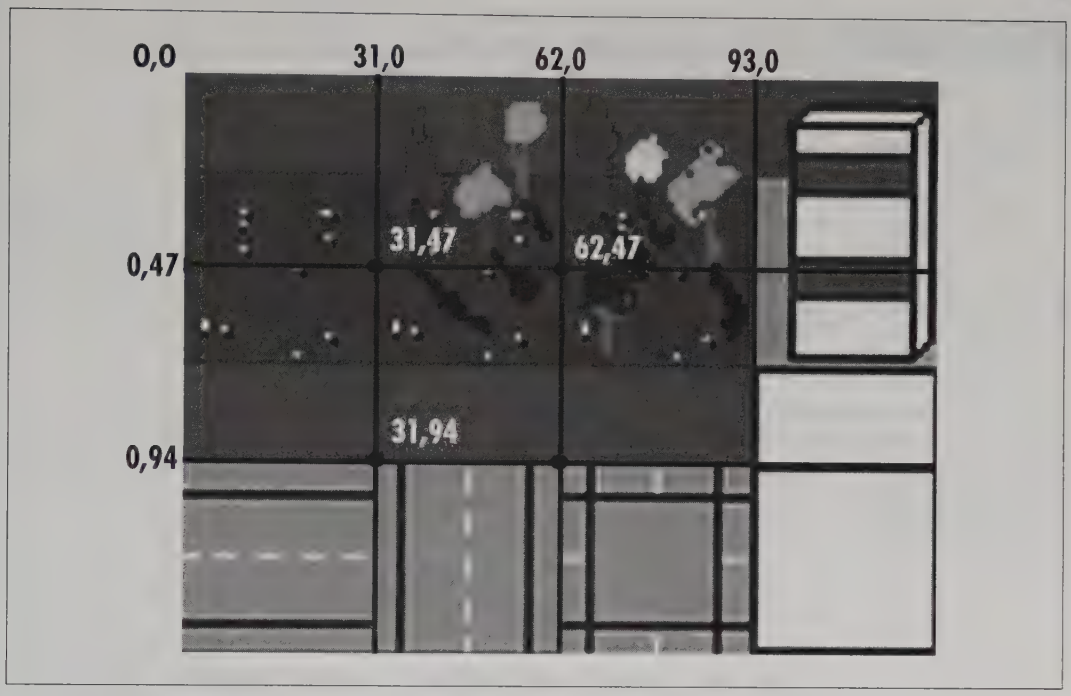


FIGURE 3-33 Coordinates of tiles in lpDDSOne offscreen surface

LISTING 3-37 Defining the terrain map

```
SPOT  terrain[4][4] = {
  0, 0, 0, 3,
  0, 2, 2, 2,
  0, 0, 2, 1,
  4, 4, 4, 4
};
```

Although your actual terrain map is 60 x 60, this 4 x 4 array of struct SPOT will serve its purpose. The integer 0 in the map is the index to cells[0] which is your tile for grass. At terrain[0][3] you have the integer 3. This is the index to cells[3], which is the tile attribute for a building. The bottom row of the terrain map array contains all fours. These tiles make a road in your map. The corresponding tiles to cells are given above in the declaration of cells[].

The actual terrain map is stored in a file called TERRAIN.INF. The contents of TERRAIN.INF look very similar to your 4 x 4 terrain map, but it is a 60 x 60 terrain map. The code for retrieving the map and initializing the terrain map is found in source file PWRGAME.CPP, and is shown in Listing 3-38.

LISTING 3-38 Initializing the terrain map

```
BOOL initMap()
{
  FILE *stream;
  int i,j;
  // Open the terrain.inf file for reading.
  if(!(stream = fopen( "TERRAIN.INF", "r")))
    return FALSE;

  // Go through the file and initialize the terrain array.
  for (i = 0; i < TERRAIN_HEIGHT; i++)
    for(j = 0; j < TERRAIN_WIDTH; j++)
    {
        fscanf(stream,"%d",&terrain[i][j].cell);
        terrain[i][j].suitPtr = NULL;
    }
  fclose(stream);
  return TRUE;
}
```

The code basically opens a file stream, then executes a nested loop to walk the cells of the terrain map and load their respective values. The code does essentially two things of importance here. For one, you are initializing the terrain map with tile IDs using standard C library call fscanf(). Second, the terrain map also contains information during gameplay if a PowerSuit is standing in that cell. When you initialize the map there are no PowerSuits, so you initialize all the terrain map PowerSuit pointers to NULL.



Opening a Window into the Virtual World

The terrain map is read in from a file in system memory. The terrain map consists of tile IDs and initialized NULL pointers of PowerSuits. They're not standing in a cell yet. Your viewport is described by 15 x 15 cells of 32 x 32 pixels (viewport is 480 x 480). You have a terrain that measures 60 x 60 cells. You want to generate the view for the viewport based on where a PowerSuit is standing. When the game initializes there are two PowerSuit teams. Each team consists of one team member. The N key on the keyboard toggles between the two players. When you toggle you want to generate a view based on where the currently selected PowerSuit is standing. Preferably you would like to generate a view with the PowerSuit standing dead center of the viewport. However, there are some special cases where this isn't always possible and I'll discuss these a little later. At the moment though, what's more important is generating the viewport display. To generate the display you need to know the *x* and *y* coordinates of the PowerSuit that is currently selected. The PowerSuit has a special data structure with fields for which cell it is standing in on the terrain

map. These *x* and *y* values are assigned to values *mapPosx* and *mapPosy* respectively. Using these two values you will determine where on the map terrain the top-left corner of your viewport is located. The values *mapPosx* and *mapPosy* are bounded by the physical dimensions of the terrain map, *terrain[TERRAIN_HEIGHT][TERRAIN_WIDTH]*, which is 60 x 60. From this you have the following:

```
mapPosy = pwrCurrent->Posy - 7;
mapPosx = pwrCurrent->Posx - 7;
```

Based on the current position of the PowerSuit, you obtain the values for the upper-left-hand corner of the viewport. The viewport measures 15 x 15 cells so you need to ensure that based on these *mapPosx* and *mapPosy* coordinates you can draw a 15 x 15 viewport. If the PowerSuit is standing in *terrain[59][59]* then obviously you can't center justify the viewport. You must adjust the viewport *mapPosx* and *mapPosy* coordinates so you can draw all 15 x 15 cells. You do this with a check to make sure your *mapPosx* and *mapPosy* coordinates leave you with enough room to draw your viewport without going outside of the terrain map bounds (as would be the case with a PowerSuit in *terrain[59][59]*). You check and perform the following operations accordingly, for the *mapPosx* value:

```
if (mapPosx < 0)
   mapPosx = 0;
if (mapPosx > TERRAIN_WIDTH-15)
  mapPosx = TERRAIN_WIDTH-15;
```

If *mapPosx*, the upper left corner of the viewport, shown in Figure 3-34, is less than 0, it is out of view bounds. You must adjust the viewport back to *mapPosx* = 0. Likewise, if *mapPosx* is greater than 60, then you must shift it back

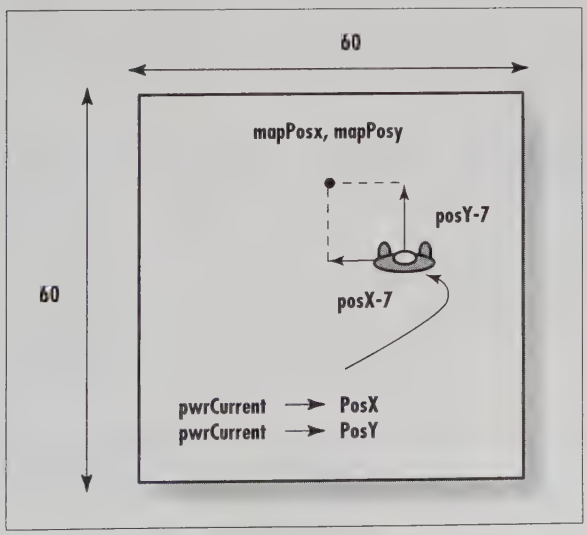


FIGURE 3-34
Determining the upper left corner of the viewport

to a value within bounds. For the *mapPosy* values of the upper left corner you perform similar operations:

```
if (mapPosy > TERRAIN_HEIGHT-15)
   mapPosy = TERRAIN_HEIGHT-15;
if (mapPosy < 0)
  mapPosy = 0;
```

If *mapPosy* is above or below the map terrain bounds, you must adjust them. These operations ensure that no matter where the selected PowerSuit is standing in the terrain map, you generate a view based on their location. The first step to generate the viewport is for the PowerSuit to be standing in the center. If this isn't possible, then you adjust the upper left values of the viewport to the best guess match.

Once the viewport has its *mapPosx* and *mapPosy* values, you can examine the terrain map to see what tiles are located in this 15 x 15 piece of real estate. The tiles in this 15 x 15 region are what's going to be drawn to the display. Retrieving the tile data requires a nested loop that looks like this:

```
for OUTER LOOP walk viewport rows
  for INNER LOOP walk each cell of row
    get tile from terrain map
```

The viewport is a two-dimensional grid measuring 15 x 15 cells (recall it's 480 x 480). The nested loop walks the viewport grid and retrieves the tile values from the terrain map that are to be displayed. This code, from source file PWRSCRN.CPP routine drawCells(), is shown in Figure 3-39.

LISTING 3-39 Walking the terrain map

```
// The actual Y offset where to put the current row of cells.
 down =TERRAIN_HEIGHT;
 left = 0;
 cellnum = 15;  // 15 cells per row in the viewport
 // Outer loop of drawing cells.
 for (j = 0; j < 15; j++)
{
   for (i = 0; i < cellnum+1; i++)
  {
    // Now reference the terrain array and get what kind of cell it
    // holds then take that information and reference the cells array
    // to get the source rectangle.
     rcRect.left = cells[terrain[j+mapPosy][i+mapPosx].cell].posx;
     rcRect.top = cells[terrain[j+mapPosy][i+mapPosx].cell].posy;
     rcRect.right = cells[terrain[j+mapPosy][i+mapPosx].cell].posx
                          + cells[terrain[j+mapPosy][i+mapPosx].cell]
                           ➥.width;
     rcRect.bottom = cells[terrain[j+mapPosy][i+mapPosx].cell].posy+cells
                          [terrain[j+mapPosy][i+mapPosx].cell].height;
     tempdown = down - cells[terrain[j+mapPosy][i+mapPosx].cell].height;
    if (tempdown < 0)
```




```
    {
       rcRect.top = cells[terrain[j+mapPosy][i+mapPosx].cell].posy -
➥tempdown;
       tempdown = 0;
    }
    // Now draw the cell.
    drawOneToBack(left, tempdown, rcRect, FALSE);
     //
     // draw the PowerSuit if one is in cell
     //
    left += 30;
   }   // for (i = 0...
  down += 30;
  left = 0;
}   // for (j = 0...
```

The primary operation being performed in the loops is determining where to blit the tile from surface *lpDDSOne*, the offscreen surface, to the flipping surface *lpDDSBack*. The retrieval of the tile ID looks pretty complicated, but it isn't. Remember that the terrain map is a 60 x 60 lookup table containing cell IDs and possibly a pointer to a PowerSuit if one is there. What you're after in these loops are the tile IDs. The tile ID is essentially the index into the *cells[]* array. Remember this array contains the attributes of the tile. Obviously, the attribute you are after here is where the tile is stored in offscreen surface *lpDDSOne*. You compose a source RECT for blitting from these attributes. The destinations *x* and *y* for the blit to flipping surface *lpDDSBack* are composed from the computations of the variables *tempdown* and *left*.

With each tile you retrieve for your 15 x 15 viewport, you blit it to the flipping surface *lpDDSBack*. This is a different approach from the DIB PowerSuit where you wouldn't even consider updating the entire screen on the fly. The work of all the blitting is handled by a routine in PWRSCRN.CPP called drawOneToBack(). This is a modularized routine of DDTOOLS.CPP function BlitSurface(). This routine is shown in Listing 3-40.

LISTING 3-40 Drawing tiles from lpDDSOne to lpDDSBack

```
int drawOneToBack(int x,int y, RECT rcRect, BOOL Full)
{

    HRESULT             ddrval = DD_OK;
    DWORD               dFlags;

    if(Full)
        dFlags = DDBLTFAST_WAIT | DDBLTFAST_NOCOLORKEY;
    else
        dFlags = DDBLTFAST_WAIT | DDBLTFAST_SRCCOLORKEY;

    return (int) BlitSurface( lpDDSOne,lpDDSBack, x, y,&rcRect, dFlags);
}
```

These routines are the primary working bodies of code for generating the dynamic viewport in *Super PowerSuit*. The source code for *Super PowerSuit* contains some intermediate-level code and will be discussed throughout this book. The *Super PowerSuit* game is designed to be hacked at. For those of you comfortable with data structures, go play with the PowerSuit linked list. As I have hinted at, it is designed to handle more than one PowerSuit per team. That's why Mike Thiels went through the trouble of coding special pointers to cells on the physical map. You can also play with the animation cells of the PowerSuits to give them smoother animation. Think of it as *Fisher Price* coding. You gave yourself a big plastic hammer to bang away at some code. If you screw up, just erase the directory and copy it back in from the CD-ROM. Have some fun, and hack away at it.

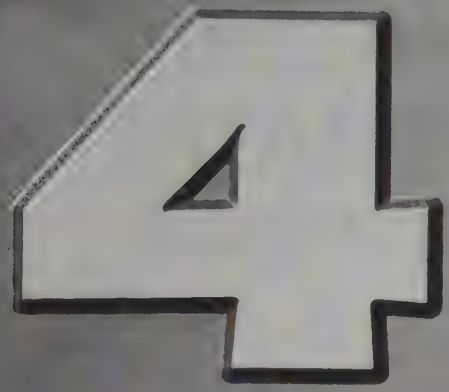


Direct Enlightenment

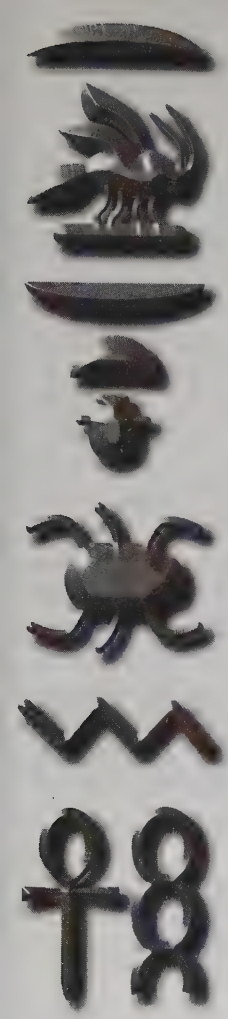
By now you should be *Direct-ly enlightened* by the power of the Microsoft Game SDK DirectDraw environment. It's hard to believe frame-rates made a quantum leap from 30 fps in DOS, to 60, and even 90 frames per second with DirectDraw. My personal enlightenment happened when I watched a demonstration of *Wing Commander III* with live video animation get 90 frames per second just by being ported over to DirectDraw. But you don't need any convincing. You probably have your own incredible story to tell, which is why you're studying the features of the Windows 95 Game SDK. But just remember, you're in it for the games, man, and the pursuit of better gameplay. But then again a steady paycheck does keep your spouse happy, too.



THE SECRETS OF THE ARTIFACTS



DIRECTINPUT: INTERFACING THE ARTIFACT



DIRECTINPUT: INTERFACING THE ARTIFACT

Designing the user interface is one of the single most difficult tasks in game design. The user interface includes the viewport display, menus, dialogs, and every other component that requires user interaction for your game. One of the key factors in designing the user interface is how well it draws the player into the game. In today's world, where there is a vast gaming hardware aftermarket, the game player has already selected a weapon of choice, whether it be a game control pad, a multi-axis digital joystick, or just the mouse that came with the system. Processing user input and responding to user actions is a crucial mechanism for drawing players into a game.

In the pages that follow, you'll examine the different ways the PC processes user input. This will include how to work with a message-driven environment. You'll also work on an input driver program to experiment with different ways Windows 95 handles device input. And of course, you'll work with the DirectInput API to interface analog and digital joysticks to your game system.

EPISODE INTERFACING WITH THE ARTIFACT



Now wasn't such a good time to ask 13 if he had ever tried this experiment on prisoners first. The PowerSuit 13 designed was fairly comfortable. The miles of sensory wires connected to my body were a little awkward. Wires and ribbon cables crossed all over the suit, which left me with the impression that this was 13's prototype, and most likely untested.

"Lie back and relax," 13 requested as he laid me down on a sliding table. The table was connected to a huge device that resembled a magnetic resonance

imaging system used in hospitals. 13 paused for a moment before he finished the final connection.

"Remember, Bary, you promised to come back and teach me to surf." 13 smiled for a moment.

"Don't you worry, I'm going in after a friend. When I find her, I'm coming back for you. Then it's Waikiki, *you and me, pal*, surfing some tubes, dude."

13 finished sliding me into the magnetic resonator. The opening was just large enough to allow the top of my head in. Inside, it was dark. I could feel some sort of mask, which was cold, lowering down, and form fitting to my face.

"So far so good, Bary. I am activating the human I/O hardware. You'll see two green beams of light. These are extremely low-powered carbon dioxide lasers. The only way to interface to the human brain without surgery is through the eyes. The back of the human eye is the only place the human brain is directly exposed. I can essentially program imagery to your brain with packets of light. My supercomputer will process responses from the data collected with the magnetic resonance imaging system. I am transmitting a test pattern now."

"Hey, that's cool, 13. It's the monster from *The Creature from the Black Lagoon*. I feel like I'm standing in the middle of a 1950s B movie," I responded with great enthusiasm.

"Technically, Bary, you are. It's the 3D test pattern I've been using for my experiments," 13 reported. "On your mark, I'm ready to interface you to the artifact."

"Whoa, wait," I interrupted. "Even the Grinch is going to see the power surge from the resonator and trace the drain to here."

"I ironed out that little problem months ago," 13 replied. "I am ready on your mark."

In my best attempt to imitate Captain Picard, I used the standard Federation order. "*Engage*!"

Nothing. There was darkness.

An image began to form, but it was hard to make out what the image was. It looked a lot like when an operating system has the wrong video device driver, and the video monitor is out of synch. Then the picture became clear.

It appeared to be a tomb or crypt entrance, but part of it was in millions of colors of resolution, and a third of it was in monochrome or not there at all.

"Bary, I've your visuals on my monitor. That tomb appears to be Agamemnon's virtual entry port. I don't know why the image is all messed up. It appears the image, from the artifact's side, is corrupted. Possibly bad memory."

"It's far worse than that, 13. That's the Zero Virus attacking this area of memory. Dr. Goldmen was here. We must hurry. This port is unstable. You may have to find another way out for me once I'm inside."

I continued in through the artifact's port. In this virtual world it resembled the entrance to a tomb. Inside the portal was an Egyptian woman. She was pale in complexion, as if she wasn't alive.

"I am Isis. Guardian of the port. Are you a lost soul or a god?" She hissed and showed fangs.

"I am a god. I travel the universe consuming worlds. My herald has come here before me," I bluffed.

"Is it you that has caused this to my world?" The vampiric guardian pointed to a wall that was crumbling from the devastation of the Zero Virus.

"I have done this to show you my power. Now let me pass or I will destroy you!" I spoke rather omnipotently.

"Spare me. You may enter through my gates." The creature slithered the words off her tongue.

"You consume worlds?" 13 questioned as I crossed the portal into the virtual nether world.

"13, there are three things that are important in life: surfing and comic books," I answered, full of myself.

"That's two, Bary. What's the third?"

"I'll point that out to you on the beach when we go surfing."



Operating Systems and User Input

It was just last year; I remember working on a project using Watcom C++ and DOS4/GW to put together a game demo for a protected-mode environment. Extended DOS was the way of the future. As a developer, if I was to survive, I would have to port all my device drivers over to a 32-bit flat memory architecture and prepare for this new frontier in programming. No sooner had I finished my software port to 32-bit DOS extenders than Microsoft released the Game SDK beta 1.0. It only took a few minutes of playing with a Game SDK demo for me to realize the winds of change were again upon my sails. That afternoon, I went home with a copy of the Microsoft Game SDK and I archived all my DOS extender libraries for keyboard and joystick input.



Real Mode Input Processing

The PC architecture has changed drastically over the past decade. Interestingly, there has been one constant in the hardware's unstoppable ebb of change: its operating system, MS-DOS. The first release of MS-DOS, MS-DOS 1.0, back in 1981, ran on a 4 MHz IBM PC. In 1984, MS-DOS 3.0 was shipped, which is most notably the DOS standard for IBM PC 80x86 architectures. Today, MS-DOS is at release 6.22 and is still a very popular operating system that will run on 200 MHz Pentium PCs. Even with Microsoft making a strong effort to replace MS-DOS, its relatively small memory requirements, hardware compatibility, and large install base make it popular with many users and publishers.

The first PCs to use MS-DOS used Intel 8086 processors with speeds in the range of 3.77-8 MHz. The 8086 processor is capable of addressing 1 MB of memory using segments and offsets. A segment is a selected area of computer memory that can be up to 64 KB in size. The offset is a 16-bit value representing the distance from the start of the memory segment.

To address memory in the range of 00000 - FFFFFh (0 to 1,048,575), the 8086 architecture uses 20-bit addressing called physical addresses to uniquely reference a byte in memory. The processor computes the 20-bit address by taking the contents of a segment register (there are four segment registers: code, data, stack, and extra) and shifting this 16-bit value four times to multiply by 10h. This operation in base 16 has the same significance as if you multiplied a base 10 value by 10. You pick up an extra zero digit in base 16. This digit is an implied digit. That is, the processor can only hold a 16-bit value in any of its four segment registers. However, if this particular digit is always 0, you can implicitly pick up 4 extra bits. Adding the 16-bit offset, you can address up to 1 million bytes of memory.

This memory architecture in the Intel family of 80x86 processors is called real mode addressing. MS-DOS is a real mode operating system that provides a system of services such as memory management, disk and file management, device input/output, and other program managers. Programs executing under MS-DOS can access system functions by loading registers with requests and function-specific information. The system function request is then sent to the operating system using a software interrupt. Likewise, the hardware can communicate with the processor using interrupt signals to suspend the current processor activity. When the processor task is suspended, a program called an interrupt handler takes action on the signal and then returns control of the processor to the suspended task.



Interrupts

On the PC, hardware communicates to the processor via interrupts. When a user presses the 6 key on the numeric keypad to move a character in a video game to the right, the keyboard sends a signal to the interrupt controller. The interrupt controller receives signals from all input devices connected to the PC. When the interrupt controller receives the signal that the numeric key 6 has been pressed, it sends a signal to the processor. At the same time, it places the numeric identification of the device requesting service out on the data bus. The processor checks the interrupt line from the interrupt controller before it fetches its next instruction. If there is no signal, the processor proceeds with its next instruction fetch. At this moment, the processor checks the interrupt line and sees that the interrupt controller is signaling. The signal sent by the interrupt controller is called an external interrupt, and it notifies the processor by setting the interrupt flag (IF). When the processor sees this flag, the contents of the flag register, the addresses in the code segment (CS) register, and the

instruction pointer register (IP) are pushed on the stack. This process saves the addresses of the instructions the processor was about to handle before the interrupt flag was set. The 80x86 then transfers processor control over to the interrupt handler routine. Figure 4-1 shows the interrupt handling process in a PC when a key is pressed.

The operating system on the PC understands various means of processing input. Each routine for handling input belongs to a category of input devices. These routines are part of the operating system and are resident in memory.

Interrupt Vector Table

Addresses to the memory resident input-handling routines are stored in the interrupt vector table. In real mode, the first 1024 bytes of memory are reserved for the interrupt vector table. The table contains 4-byte entries, consisting of an offset and a segment. These 4-byte (32-bit) entries are the addresses to the interrupt handler. Interrupt handlers are the tasks called to process a specific interrupt. The first entry in the table, interrupt 0, is at absolute address 0. The

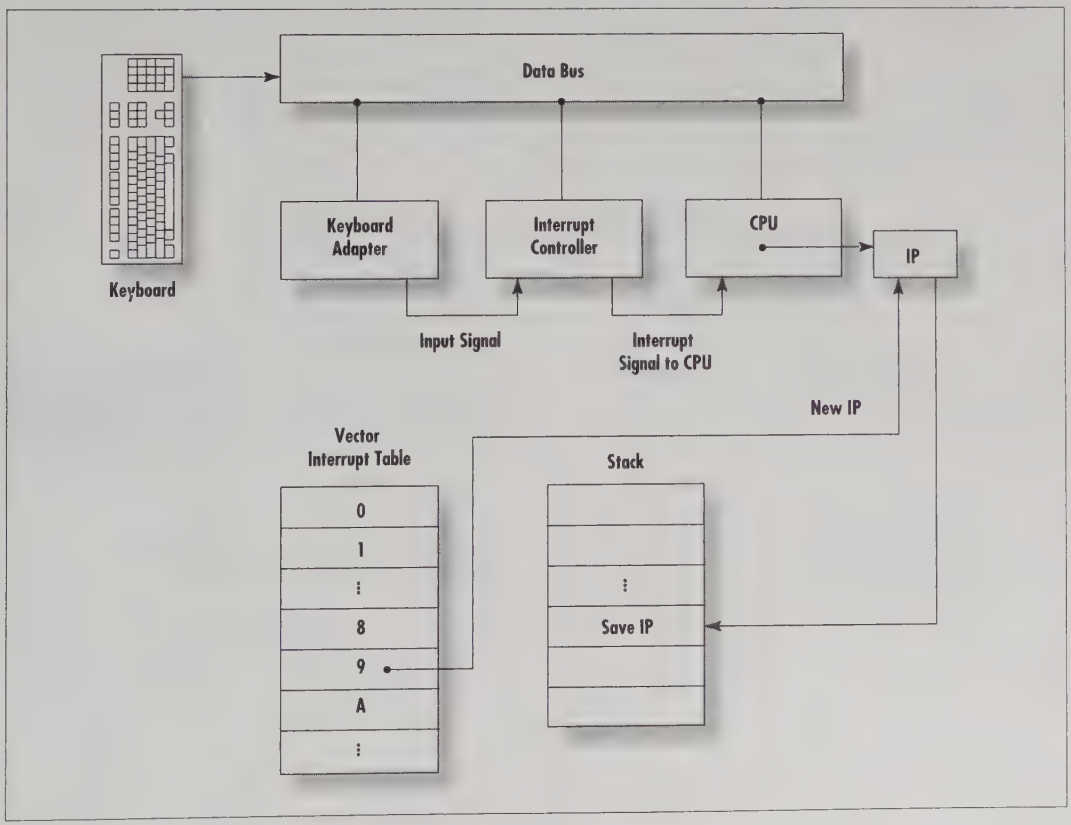


FIGURE 4-1 Processing a keystroke

second interrupt, interrupt 1, is at absolute address 4. The absolute vector address is the vector number times four. The address for the keyboard handler, interrupt 9, is at 9 x 4 = 36. When the processor sees an interrupt signal from the keyboard, the processor knows it's a keyboard request because the interrupt number 9 came in over the data bus. The processor suspends its activities and relinquishes control to the routine whose address is at absolute memory location 36 (or entry 9 in the interrupt vector table). The vector interrupt table has 1024 bytes reserved for its entries. Each entry is 4 bytes long, as shown in Figure 4-2. The 8086 can store up to 256 entries in the table.

When the interrupt handler is finished, it returns control to the processor, which resumes with the task it suspended. The instruction IRET is used to resume the processor task execution. The IRET pops the flag register and the CS and IP registers back into their respective registers in the processor. Control by the processor has now been resumed.



DOS Interrupts and Functions

When the PC is booted, primitive device drivers and system startup programs, located in BIOS read-only memory (ROM), are read into RAM during system initialization. The BIOS supports low-level functionality that is hardware-dependent and not available in DOS. Higher functionality, such as disk I/O, standard text display output, and other I/O are handled by DOS interrupts.

Memory Address	Interrupt Funct
0 – 3	0 Divide by Zero
4 – 7	1 Single Step
8 – B	2 Nonmaskable Interrupt
⋮	⋮
24 – 27	9 Keyboard Interrupt
⋮	⋮
58 – 5B	16 Keyboard Interrupt
⋮	⋮
84 – 87	21 DOS Function call

FIGURE 4-2
The PC interrupt vector table

Interrupt 21h is reserved for DOS function requests. Although DOS interrupts have interrupt vector entries 20h - 3Fh, interrupt 21h is the most commonly used by programs. The convention for calling DOS functions is to place the desired function in the AH register and call interrupt 21h (INT 21h). DOS functions preserve all the registers in the processor except for the ones that will return values from the interrupt to the program.



Keyboard Input Under DOS

When a user presses a key to move the character on the screen, the keystroke is saved in a 15-character typeahead buffer. The BIOS keyboard interrupt is responsible for placing it there. Programs can examine this keyboard buffer using DOS functions to return the next available character in the buffer.

DOS functions have their limitations, especially when it comes to buffered input. For instance, using DOS functions for keyboard input, DOS may wait until a return key is pressed before sending the input buffer to the calling program. If the user fills up the buffer and enters one character less than the maximum buffer number, the familiar keyboard squawk will sound.

One way to overcome the limitations with DOS in a game application is to bypass MS-DOS. A hardware-dependent technique is to use ROM BIOS keyboard interrupt 16h. INT 16h reads a single character from the keyboard input buffer and returns it in the AL register. BIOS routines offer more flexibility and are capable of returning keycodes that are not typically returned by MS-DOS. BIOS functions can tell you if a key is waiting and the keyboard Shift states.

BIOS routines are supported by most MS-DOS compilers. Using the Watcom C++ compiler example (Visual C++ 32-bit DOS development), you can use the code in Listing 4-1 to see if NumLock is pressed.

LISTING 4-1 Checking the state of the NumLock key using DOS4/GW

```
#include <stdio.h>
#include <bios.h>

#define NUMLOCK_KEY 0x20
void main()
{
  unsigned short keyState;

  keyState = _bios_keybrd( _KEYBRD_SHIFTSTATUS );
  if( keyState & NUMLOCK_KEY )
  {
    // handle numlock key down
  }
}
```

The Watcom C++ run-time library supports INT 16h BIOS keyboard services with the call to _bios_keybrd(). Using BIOS-supported routines has its advantages and disadvantages. Many game developers feel that even BIOS keyboard services are too slow and have developed their own keyboard drivers. The fact that MS-DOS is a real mode operating system gives programmers the flexibility to go in at low levels and make adjustments to program flow at all levels of their software.



Protected Mode Input Processing

The greatest advantage of real mode programming is the flexibility in manipulating the entire flow of your application, almost down to the DOS kernel level. The disadvantage of real mode is the memory constraints. You can only address up to 1 MB using MS-DOS and, of it, only about 640 KB is available to the application. The processors after the 8086, the 80286 through the 80486 and Pentiums, are capable of accessing memory in the 16 MB range for the 80286 to 4 GB for the 80386, 80486, and Pentiums (see Intel 80x86 Programmer's Reference Manuals). That's the good news. The bad news is the non-systems programmer has been removed from the low levels of the operating system in an architecture called protected mode. Protected-mode operating systems, called DOS extenders, are available for MS-DOS. Windows 95 is a protected mode operating system. The memory architecture under protected mode is flat, which means you can address 4 million bytes in memory without having to use a segment and offset scheme real mode used for addressing.

Interrupts in protected mode are handled in a different fashion than in real mode. Real mode has the interrupt vector table to look up the vector to a device-requesting service. In protected mode, things can be a little more complicated and more than one device can be requesting an interrupt service. Also, more than one application can be executing at once.

Protected mode uses an interrupt descriptor table (IDT) that contains up to 256 descriptors (pointers) to interrupt service routines. Every interrupt used by the system must have an entry ID in the IDT. The organization of the IDT is nothing like the interrupt vector table. A descriptor table references locations in memory where interrupt service routines can be found. The service routine's addresses are not stored in a logical 4-byte order as they were with the vector table. In protected mode, the interrupt vectors are 8-byte values stored in the IDT, as shown in Figure 4-3.

The operating system is responsible for loading the service routines into memory when it initializes at startup. At that time, the interrupt vector table contains a physical design of how and where the interrupts would be located

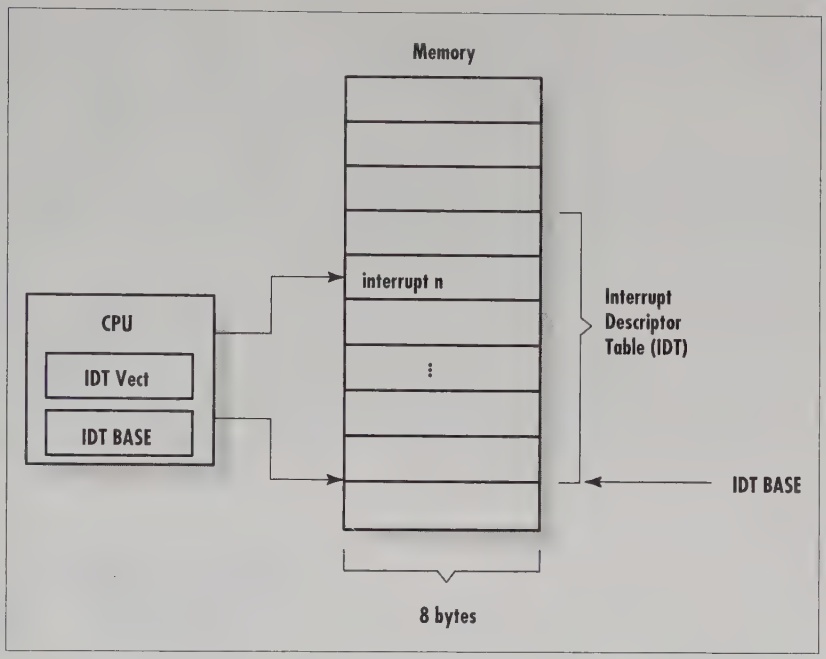


FIGURE 4-3
Protected mode interrupt descriptor table

at run-time on all machines. This is not the case with protected mode. With the use of a descriptor table, the service routines can be stored in any order. The order of storage is determined by what the operating system determines to be efficient.

Under protected mode, the operating system is more complicated because of the number of tasks the kernel may have to service. The kernel executes at protection layer 0 (PL 0) and doesn't yield privilege to any other task to tamper with its resources. Illegal tampering with system resources results in a General Protection Fault (GPF) error, which Windows users have seen on occasion. A GPF is a notification from the kernel that you have violated ring protection rules, such as attempting to access a segment that doesn't belong to your task, and that your application is being shut down.



Windows Messaging and User Input

Windows 95 is a protected-mode preemptive-multitasking operating system. This means the processor can be handed off to another process in the middle of your application's task. For that matter, the processor could be handed off to another instance of your application. In this multitasking environment, it's difficult to determine which application or instance of an application gets

which interrupt or shared resource. The complexity of resolving this is removed from the programmer and left in the hands of the Windows 95 operating system designer.

When a key is pressed, the interrupt is processed by the operating system. The operating system understands which application instance needs this keystroke. A message is sent to the application notifying the program of user input. The application then responds to the event. Figure 4-4 shows the keystroke event being relayed up through the protection levels of the Windows 95 operating system.

Under the Windows 95 operating system, the game programmer must concentrate on how to service a Windows 95 event message. Working with messages is frustrating to DOS programmers who enjoy working at a low level in the operating system. However, as you'll soon see, the concepts are the same as to how user input is handled. Just the manner in which they're handled has changed.

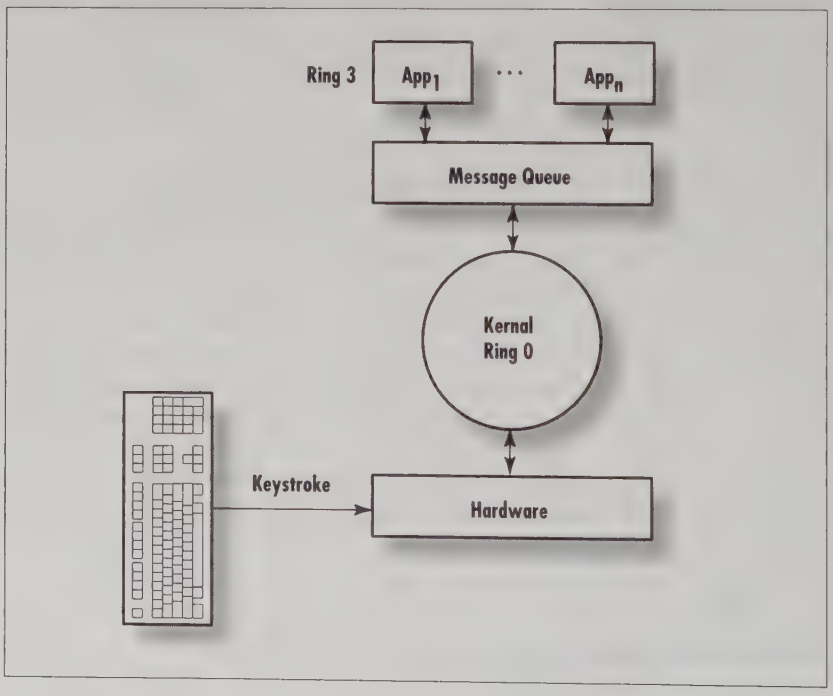


FIGURE 4-4
Keystroke event being handled by Windows 95



Handling User Input Events

In this section, you're going to put together an application that will allow you to rapidly prototype and experiment with different forms of message-based user input. Usually all you need to test a new concept is some simple art to blit and a no-frills application to drive it. The application DIDRVR, DirectInput driver, will allow you to do just that. As you'll see, it's modular so you can snap in any code you want to experiment with a new input device. Your example application will demonstrate, with the cameo assistance of Bubonic the WartHog from Chapter 9, how to integrate keyboard, mouse, and joystick movements into your application. The joystick implementation will use the Microsoft DirectInput API.



The DIDRVR Application

The purpose of DIDRVR is to create a quick and dirty application to experiment with different forms of input. I developed an application like this when I worked with DOS. This application is great for experimenting with ideas, debugging input problems where the code isn't too sophisticated, and implementing a new API. In fact, this same application was used to implement a test with an Exos force feedback joystick. The integration went rather easily using this as an application test template.

The DIDRVR application consists of your Generic Windows 95 Game Application from Chapter 1, CDib from Chapter 2, DDTOOLS from Chapter 3, and some new code to handle the input. Snapping these modules together, you can construct your DirectInput driver application.



DIDRVR Animation

The DIDRVR test application uses simple animation to test the program's input-handling routines. Here you are going to use a few cells of animation to run Bubonic the WartHog to both sides of the screen. You will use tiles to draw a simple terrain for Bubonic to run on.

Begin construction of DIDRVR by modifying a copy of the source file GENX.CPP, which is the generic game application. To this, add the code to initialize the CDib object, which holds the .BMP data, and add the inits for DDTOOLS. This code is shown in Listing 4-2.

LISTING 4-2 Initializing the DIDRVR application

```
BOOL InitDITest(HWND hwnd)
{
  // new Dib object
  if ((pDib = new CDib) == NULL)
  {
    MessageBox(hwnd,"CDib* Allocation Exhausted","Error", MB_OK);
    return FALSE;
  }

  // read in the BMP file contents
  if ((pDib->ReadFile("ditest.bmp")) == NULL)
  {
    MessageBox(hwnd,"CDib* Read BMP Failed", "Error", MB_OK);
    return FALSE;
  }

// initialize the DirectDraw object
if (DDInit(hwnd) == FALSE)
{
  MessageBox(hwnd,"DirectDraw Init Failed", "Error", MB_OK);
  return FALSE;
}

  // initialize the DirectDraw Flip Surfaces
  if (DDSInit() == FALSE)
  {
    MessageBox(hwnd,"DirectDraw Flip Surface Init Failed", "Error", MB_OK);
    return FALSE;
  }

  // erase the flip surfaces
  ErasePrimarySurface();
  EraseBackSurface();

  // copy the RGBQuad DIB palette
  SetRGBQuadPalette(&lpDDPal, pDib->GetLPRGBPtr() );

  // did we get a palette?
  if (lpDDPal == NULL)
  {
    MessageBox(hwnd,"IDirectDrawPalette* NULL", "Error", MB_OK);
    return FALSE;
  }

  // set the Primary surface palette
  lpDDSPrimary->SetPalette( lpDDPal );
  lpDDSBack->SetPalette( lpDDPal );

  lpDDSOffscreen = GetNewSurface( pDib->GetCDibBiWidth(),
                                  pDib->GetCDibBiHeight() );
  // did we get a surface?
```

```
  if (lpDDSOffscreen == NULL)
  {
    MessageBox(hwnd,"lpDDSOffscreen NULL", "Error", MB_OK);
    return FALSE;
  }

  // copy the Dib object bits to an offscreen surface
  if (CopyDibToSurface( lpDDSOffscreen, pDib ) == FALSE)
  {
    MessageBox(hwnd,"CopyDibToSurface Failed", "Error", MB_OK);
    return FALSE;
  }

  SetTransColorKey( lpDDSOffscreen, 9, 92);
  // initialize the wart hog
  wartHog.direction = HOG_GO_RIGHT;  // facing to the right
  wartHog.currentCell = 0;  // first cell of stop animation
  wartHog.posx = 32;
  wartHog.posy = 298;
  wartHog.delay = 90;
  wartHog.lastTickCount = GetTickCount();
  return TRUE;
}
```

The initialization steps of function InitDITest() are broken down graphically in Figure 4-5. For a simple bare bones test application, your initialization involves setting up the resources necessary for animation. You have no idea what kind of input drivers or APIs you are going to use at this point. The main feature you are aiming for is modularity to snap anything you want into this application.

The initialization code can be broken down into four steps. The first step is to create a CDib object to hold your .BMP file data. Then read the .BMP file into the CDib object. The image in CDib object is your sprites of Bubonic and the terrain tiles you will use. In Step 2, create the DirectDraw object to use the fast hardware-drawing routines of the Microsoft Game SDK. If the DirectDraw object initializes properly, initialize the flipping surfaces. In Step 3, instantiate DirectDraw objects based on the CDib data. Create the DirectDrawPalette from the RGBQUAD palette in CDib object and then set the palette. In this step you also create an offscreen DirectDrawSurface based on the size and attributes of the CDib image data.



Step 1. Initializing the CDib Object

The animation cells for your input test application are stored in a .BMP file on disk. You are going to use the CDib object you designed in Chapter 2 to read in the image data from the .BMP file. Once the image is in the CDib object, you can start copying palettes and image data, using the CDib members, to instantiate the DirectDraw objects when they're initialized. Your code for CDib instantiation is shown in Listing 4-3.

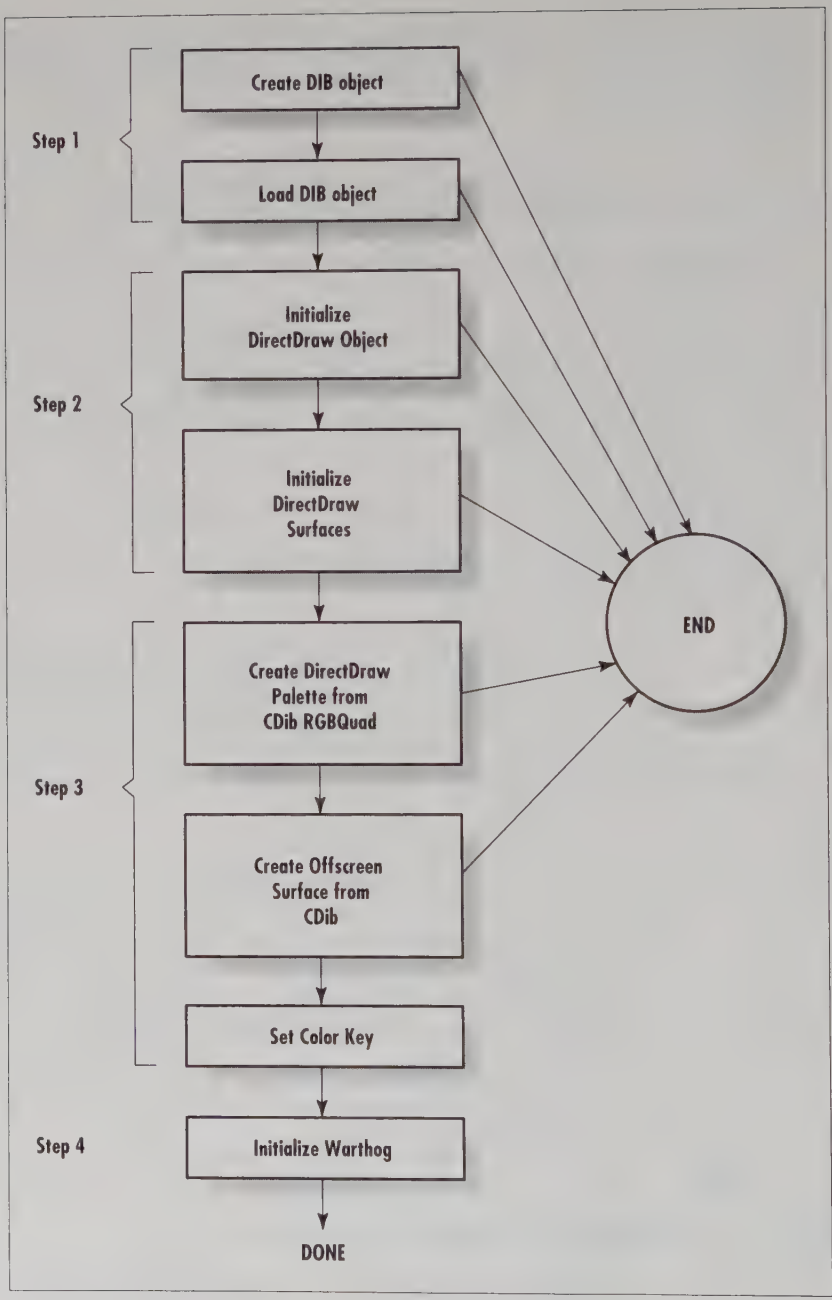


FIGURE 4-5
Initializing the DIDRVR application

LISTING 4-3 Loading the animation cells for your application

```
CDib* pDib;

// new Dib object
if ((pDib = new CDib) == NULL)
{
  MessageBox(hwnd,"CDib* Allocation Exhausted", "Error", MB_OK);
  return FALSE;
}

// read in the BMP file contents
if ((pDib->ReadFile("ditest.bmp")) == NULL)
{
  MessageBox(hwnd,"CDib* Read BMP Failed", "Error", MB_OK);
  return FALSE;
}
```

In the DIDRVR code, the CDib object *pDib is global and is declared outside of the InitDITest() routine. For this application, you could move the CDib object within the scope of InitDITest(), so it is freed automatically when the routine finishes. For this application, either way you choose is fine. If initialization of the CDib object fails, probably from memory exhaustion, then the application terminates. If you can't load your sprites, you can't test your drivers. If the CDib object fails to read the .BMP file, it also constitutes a fatal error. The image you are reading in is shown in Figure 4-6.

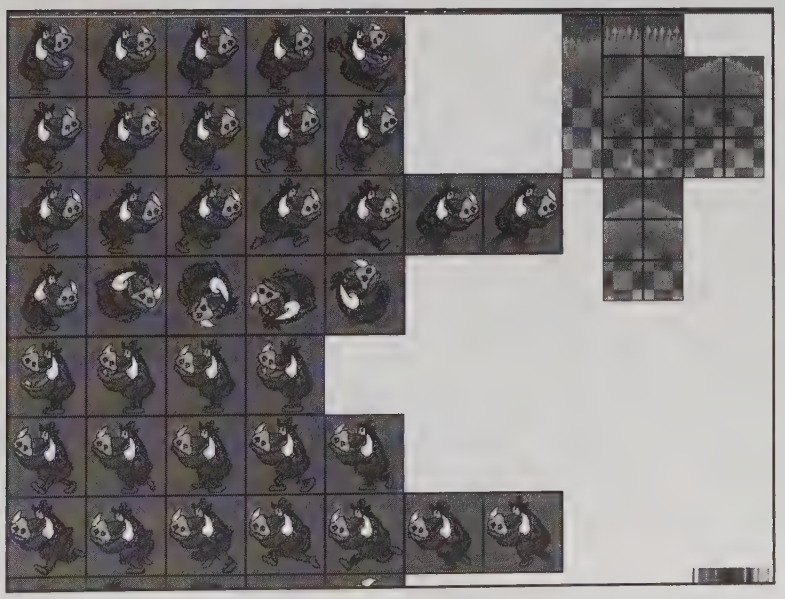


FIGURE 4-6

Sprites used for the DIDRVR application

You can modify this code to read in whatever sprites you want. As you'll see later, putting in your own .BMP file and sprites is really easy. The luxury of using the images provided is that you don't have to do the legwork of figuring all the bounding rectangles for the sprites as I have already done for you.



Step 2. Creating the DirectDraw Objects

The routine InitDITest() has successfully loaded the .BMP data into the CDib object. Now you need to create an instance of DirectDraw to utilize the blazingly fast hardware capabilities of your video cards. Using the code from DDTOOLS in Chapter 3, integrate this into your DIDRVR application as shown in Listing 4-4.

LISTING 4-4 Initialing DirectDraw in the DIDRVR application

```
if (DDInit(hwnd) == FALSE)
{
  MessageBox(hwnd,"DirectDraw Init Failed", "Error", MB_OK);
  return FALSE;
}

// initialize the DirectDraw Flip Surfaces
if (DDSInit() == FALSE)
{
MessageBox(hwnd,"DirectDraw Flip Surface Init Failed", "Error", MB_OK);
       return FALSE;
}

// erase the flip surfaces
ErasePrimarySurface();
EraseBackSurface();
```

The routine DDInit() initializes the DirectDraw object, sets the cooperative level, and sets the video mode. If you're successful and have created a DirectDraw object to communicate with your video hardware, then you can proceed to create the flipping surfaces. The routine DDSInit() initializes the DirectDrawSurfaces you will use for surface flipping. Once you have a DirectDrawSurface instance with two flip surfaces, you can erase any garbage data that may have been in there. For this, you use the two routines, ErasePrimarySurface() and EraseBackSurface(). If you're unfamiliar with the routines in DDTOOLS, they are covered extensively in Chapter 3.

Any errors in the DirectDraw object instantiation are treated as fatal. Without graphics you don't have your test program.



Step 3. DirectDraw Instantiation

This is the step where you begin assembling the various components you have already created. In system memory you have the CDib object with image data from the .BMP file. In video memory you have surfaces to use for flipping. Now you have to unload the CDib information into the DirectDraw objects. The very first thing you must do is retrieve the RGBQUAD color table from the CDib object and transfer these entries into a DirectDrawPalette. The routines copied from DDTOOLS are shown in Listing 4-5.

LISTING 4-5 Setting up the DirectDraw palette

```
// copy the RGBQuad DIB palette
SetRGBQuadPalette(&lpDDPal, pDib->GetLPRGBPtr() );

// did we get a palette?
if (lpDDPal == NULL)
{
  MessageBox(hwnd,"IDirectDrawPalette* NULL",  "Error", MB_OK);
  return FALSE;
}

// set the Primary surface palette
lpDDSPrimary->SetPalette( lpDDPal );
```

The routine SetRGBQuadPalette() creates a DirectDrawPalette object and transfers the RGBQUAD color table entries over to DirectDraw. If the call to SetRGBQuadPalette() was successful, the pointer, *lpDDPal*, should be valid. If it's not, terminate the application. With a valid DirectDrawPalette, proceed with instantiation of the DirectDrawSurface palette using the data from the DirectDrawPalette. The DirectDrawSurface member, SetPalette(), is called to write the entries in the DirectDrawSurface palette. Once you have instances of palettes, you are ready to retrieve the image bits from the CDib object.

To retrieve the CDib object image bits, you are going to need a DirectDrawSurface to store the data. You need to create an offscreen surface using DDTOOLS routines. This is shown in Listing 4-6.

LISTING 4-6 Creating the offscreen surface

```
lpDDSOffscreen = GetNewSurface( pDib->GetCDibBiWidth(),
                                pDib->GetCDibBiHeight() );

// did we get a surface?
if (lpDDSOffscreen == NULL)
{
  MessageBox(hwnd,"lpDDSOffscreen NULL", "Error", MB_OK);
  return FALSE;
}
```

continued on next page

continued from previous page

```
// copy the Dib object bits to an offscreen surface
if (CopyDibToSurface( lpDDSOffscreen, pDib ) == FALSE)
{
  MessageBox(hwnd,"CopyDibToSurface Failed", "Error", MB_OK);
  return FALSE;
}

SetTransColorKey( lpDDSOffscreen, 9, 92);
```

Using the call GetNewSurface(), create an offscreen DirectDrawSurface the size of your image bits. If the call is successful, you have an offscreen surface in video memory you can use to store your image bits. Note that if the offscreen surface initialization fails, you terminate the program. This is the surface you are going to store your sprites on.

Moving along, you have a DirectDrawSurface, *lpDDSOffscreen*, that's ready to accept image data from the CDib object. Using the DDTOOLS routine CopyDibToSurface(), begin copying CDib image data to video memory. Failure to properly copy data into the surface constitutes a fatal error.

Once the image data is in video memory, set the DirectDraw DDCOLORKEY struct to inform your blitters what value you're going to use for transparent copying. If you're unfamiliar with color transparency copying, it is discussed in Chapter 2. The DDTOOLS call SetTransColorKey() is used to retrieve the transparency pixel value from the offscreen DirectDrawSurface. Upon completion of this call, the DirectDraw object instantiations are completed and the sprites are stored in video memory, if your video hardware supports DirectDraw. Otherwise, it may be in system memory.



Step 4. Initializing the WartHog Structure

Your character, Bubonic, requires initialization to use him in your program. The functionality required of Bubonic is to walk to the left or to the right and to stop. When Bubonic stops he can be facing towards the right or left side of the screen. Listing 4-7 shows the minimal functionality required.

LISTING 4-7 Characteristics of Bubonic the WartHog for your application

```
#define MAX_STOP_CELLS 3  // 4 cells of stop animation
#define MAX_WALK_CELLS 4  // 5 cells of walking animation

enum { HOG_STOP_RIGHT, HOG_STOP_LEFT, HOG_GO_RIGHT, HOG_GO_LEFT };

typedef struct  {
  int posx;   // upper-right corner of sprite
  int posy;
  int currentCell;
  int direction;
```

```
  DWORD delay;
  DWORD lastTickCount;
}HOGTASK;

HOGTASK wartHog;

// initialize the wart hog
wartHog.direction = HOG_STOP_RIGHT;      // facing to the right
wartHog.currentCell = 0;  // first cell of stop animation
wartHog.posx = 32;
wartHog.posy = 298;
wartHog.delay = 90;
wartHog.lastTickCount = GetTickCount();
return TRUE;
```

The wartHog structure is used to update the sprite animation on the screen based on user input or screen updates for cells. When you initialize the wartHog structure, you set the wartHog direction field to HOG_STOP_RIGHT. This instructs the application that when Bubonic is placed on the screen, he is to be displayed standing and facing the right corner of the screen. There are four cells of animation to draw when Bubonic is standing. When you first render Bubonic, you want the application to start with the first cell of animation, which is 0. The wartHog structure field, *currentCell*, holds this value, and is later altered to cycle through the frames of animation.

The *posx* and *posy* fields identify the upper left corner of the Bubonic sprite when he moves around the display. The fields *posx* and *posy* contain the values 32 and 298, respectively. These are the destination coordinates of Bubonic when you first blit him to the display. Figure 4-7 shows the values of *posx* and *posy*, and how they correspond to rendering Bubonic on the primary flip surface.

The last two fields in the wartHog structure are used for updating the frames of animation. The field delay is used to set the finite time span between cell updates. You're using 486 computers or better with hardware-accelerated

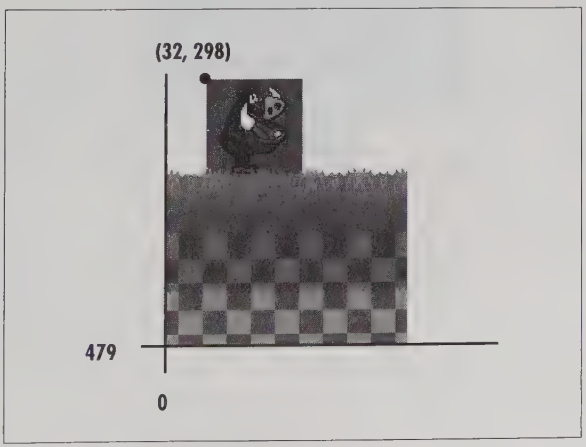


FIGURE 4-7
Initializing Bubonic for the DIDRVR application

graphics. Using a delay count maintained by the system, set the delay between animation cell changes to 90 milliseconds. On a Pentium 120 MHz system with a Diamond Stealth, or a 486 66 MHz with a Mach 64, the same frame of animation will be drawn for 90 milliseconds. When the time has expired, draw the next cell and reset the field *lastTickCount* to start the delay time all over again. The field *lastTickCount* is set by the Win32 API call GetTickCount(), which returns, in milliseconds, the elapsed time since the Windows 95 operating system was started.

The information provided by the wartHog structure is sufficient to display Bubonic when the application executes. The values of the wartHog struct will then be modified as the information is updated from user input.



The Terrain Tiles

For visual appeal, you're going to give Bubonic some grass to run back and forth on. It's really not necessary, but in a professional environment, there are just enough frills in this input test application to use it as a demo for a client or a project staff meeting. The resources are simple to add. Figure 4-8 shows the grass tile on the offscreen surface you are going to render to the primary surface.

The fields for the RECT structure are labeled in Figure 4-8. These coordinates are used to define the srcRect for blitting with DirectDraw. Since

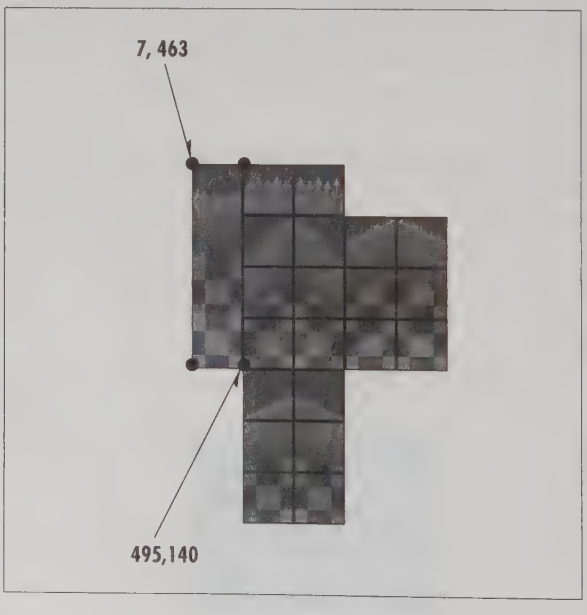


FIGURE 4-8

The terrain tile on the offscreen surface

the terrain is a tile, you will render this object 20 times across the screen to create the effect of a continuous terrain. The tile is 32 pixels wide and the surface is 640 pixels wide. Dividing 32 grass tiles into 640 pixels gives you your magic number of 20 tiles to paint the terrain. Figure 4-9 shows a graphical representation of your terrain rendering.

The terrain tile is blitted to the back flip surface from the offscreen surface during every screen update. The indices of the loop, ix, are labeled in Figure 4-9 to illustrate how the loop renders. The function BltSurface(), from DDTOOLS, is used to blit the tiles to the back flip surface. The code for drawTerrain() is shown in Listing 4-8.

LISTING 4-8 Drawing the terrain tiles

```
void drawTerrain()
{
  int ix;
  int offset = 0;
  RECT srcRect = { 463,7,495,140 };

  for (ix = 0; ix < 20; ix++)
  {
    BlitSurface( lpDDSOffscreen, lpDDSBack,offset,346,
                        &srcRect, DDBLTFAST_SRCCOLORKEY);
    offset += 32;
  }
}
```

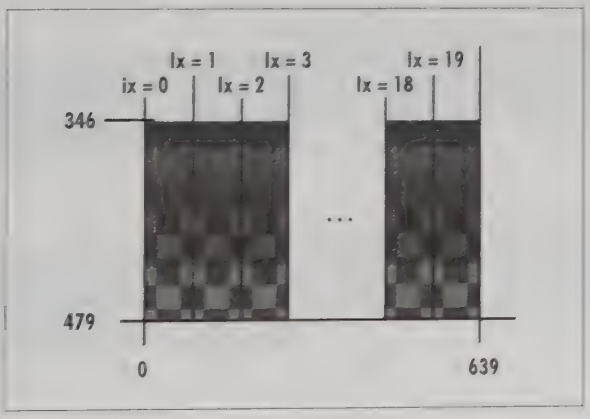


FIGURE 4-9 Rendering the terrain tiles

The variable, *offset*, is the *x* coordinate of the upper left corner of the destination RECT where the grass tile is to be rendered. The constant value 346 is the y coordinate to the upper left corner of the destination RECT. Since you're drawing a straight row of tiles horizontally you have no need for the top value to change.

Bubonic the WartHog Sprites

To animate Bubonic the WartHog, you need eight cells of animation. This isn't the smoothest of animation, but it's fine for test purposes. Bubonic has four different enumerated states used to describe what style cells you are going to use. Bubonic can be standing facing to the right or the left. There are four cells of animation for both directions. Bubonic can be walking to the right side of the screen or he can be walking towards the left. For both directions Bubonic has five cells of animation. A commercial game might have 10 to 24 frames of animation. Figure 4-10 shows the Bubonic animation sequences for standing. Animation sequences of 24 frames are typically optimum. The cells have the respective source coordinates on the offscreen surface for blitting. The cell IDs used by the *currentCell* field in the wartHog structure are also labeled.

These source coordinates are stored in an array of RECTs. The source file, DIGLBLS.CPP, contains the arrays shown in Listing 4-9.

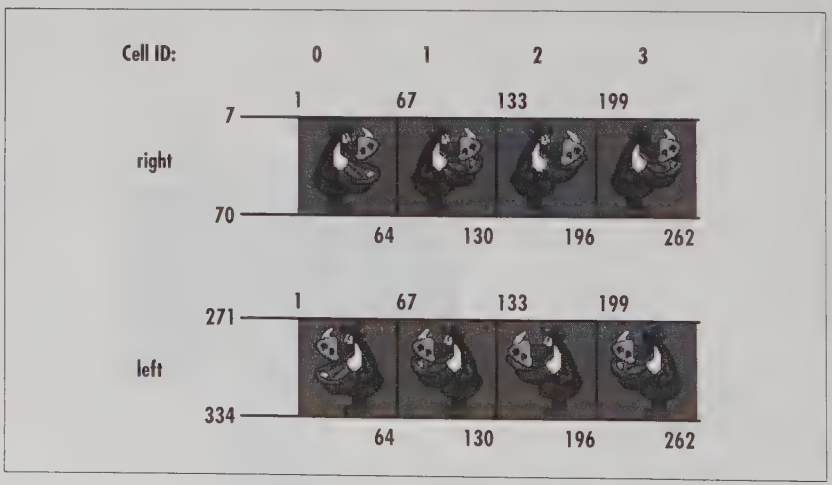


FIGURE 4-10
Bubonic animation cells for standing

LISTING 4-9 Bubonic's standing animation cell arrays

```
RECT standRightRect[] = {
  {   1,   7,  64,  70},
  {  67,   7, 130,  70},
  { 133,   7, 196,  70},
  { 199,   7, 262,  70},
  { 265,   7, 331,  70}
};

RECT standLeftRect[] = {
  {   1, 271,  64, 334},
  {  67, 271, 130, 334},
  { 133, 271, 196, 334},
  { 199, 271, 262, 334}
};
```

To animate Bubonic standing still, the wartHog structure field direction must contain the value HOG_STOP_RIGHT or HOG_STOP_LEFT. These two flags indicate state and direction for which cells to use. If the direction field contains the values HOG_GO_RIGHT or HOG_GO_LEFT, then you must use the animation cells for Bubonic walking in the desired direction. Figure 4-11 shows the animation cells and source coordinates on the offscreen surface for Bubonic walking.

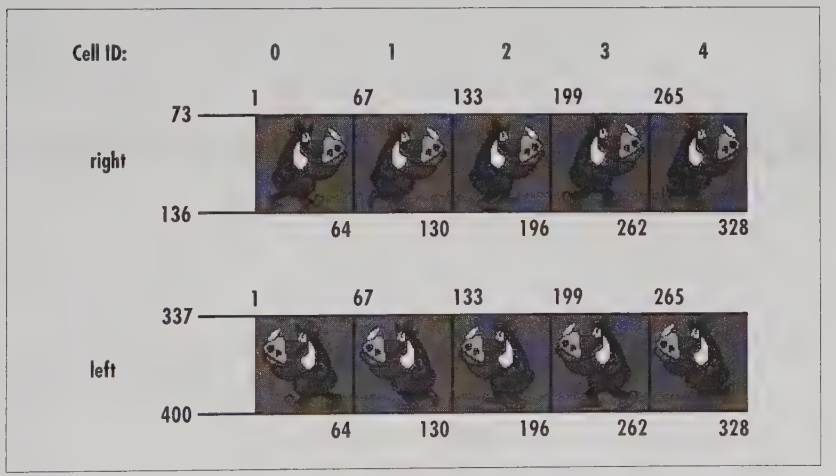


FIGURE 4-11
Animation cells for Bubonic walking

The coordinates are defined in the source file, DIGLBLS.CPP, just as you did for the Bubonic stop animation cells. Listing 4-10 provides the array of RECTs for walking animation cells.

LISTING 4-10 Bubonic's walking animation cell arrays

```
RECT goRightRect[] = {
  {   1,  73,  64, 136},
  {  67,  73, 130, 136},
  { 133,  73, 196, 136},
  { 199,  73, 262, 136},
  { 265,  73, 328, 136}
};

RECT goLeftRect[] = {
  {   1, 337,  64, 400},
  {  67, 337, 130, 400},
  { 133, 337, 196, 400},
  { 199, 337, 262, 400},
  { 265, 337, 328, 400}
};
```

The program now has sufficient information to start rendering any one of the four movement states for Bubonic. When the DIDRVR application executes, it will start with the animation sequences of Bubonic standing on the left side of the screen. At the moment, that's all Bubonic is capable of doing until he is integrated to respond to some form of user input. The code to render Bubonic in any one of his four movement states is provided in Listing 4-11.

LISTING 4-11 Drawing Bubonic in any of his motion states

```
void drawBubonic()
{
  DWORD thisTickCount;
  DWORD tickDiff;

  thisTickCount = GetTickCount();
  tickDiff = thisTickCount - wartHog.lastTickCount;


  switch(wartHog.direction)
  {
    case HOG_STOP_RIGHT:
      drawOneToBack(wartHog.posx,wartHog.posy,
                standRightRect[wartHog.currentCell], FALSE);

      if (tickDiff > wartHog.delay)
      {
        if (wartHog.currentCell++ > MAX_STOP_CELLS-1)
          wartHog.currentCell = 0;
        wartHog.lastTickCount = GetTickCount();
      }
```

```
      return;
    case HOG_STOP_LEFT:
      drawOneToBack(wartHog.posx,wartHog.posy,
                              standLeftRect[wartHog.currentCell], FALSE);

      if (tickDiff > wartHog.delay)
      {
        if (wartHog.currentCell++ > MAX_STOP_CELLS-1)
          wartHog.currentCell = 0;
        wartHog.lastTickCount = GetTickCount();
      }
      return;
    case HOG_GO_RIGHT:
      drawOneToBack(wartHog.posx,wartHog.posy,
                goRightRect[wartHog.currentCell], FALSE);

      if (tickDiff > wartHog.delay)
      {
        if (wartHog.currentCell++ > MAX_WALK_CELLS-1)
          wartHog.currentCell = 0;
        wartHog.lastTickCount = GetTickCount();
      }
      return;
    case HOG_GO_LEFT:
      drawOneToBack(wartHog.posx,wartHog.posy,
                goLeftRect[wartHog.currentCell], FALSE);

      if (tickDiff > wartHog.delay)
      {
        if (wartHog.currentCell++ > MAX_WALK_CELLS-1)
          wartHog.currentCell = 0;
        wartHog.lastTickCount = GetTickCount();
      }
      return;
    default:
      break;
              } // end switch
}
```

The function drawBubonic() is designed so that some external function directly responsible for processing user input updates the wartHog struct. Based on these changes, the function drawBubonic() draws the proper cells of animation to reflect the user response. A switch statement is used to examine which movement state the user has selected. The code in this function is independent of the device used for input or any API that may be used in conjunction with this application.



Case HOG_STOP_RIGHT:

If the user selects for Bubonic to stop moving in the right direction, the wartHog structure field direction is set to HOG_STOP_RIGHT. Listing 4-12 provides the code for Bubonic's stop-facing-right animation sequences.

LISTING 4-12 Stopping Bubonic with facing HOG_STOP_RIGHT

```
case HOG_STOP_RIGHT:
drawOneToBack(wartHog.posx,wartHog.posy,
       standRightRect[wartHog.currentCell], FALSE);

if (tickDiff > wartHog.delay)
{
  if (wartHog.currentCell++ > MAX_STOP_CELLS-1)
    wartHog.currentCell = 0;
  wartHog.lastTickCount = GetTickCount();
}
return;
```

Two events occur in this code. One, Bubonic's current animation cell is blitted from the offscreen surface to the back flip surface. The current cell is continuously used for blitting operations for a duration of 90 milliseconds. Without this delay, Bubonic would cycle through his animation sequences faster than one could comprehend what was going on. Since this is your example application, play around with the delay field in the wartHog structure; try values of 0, 30, 50, and then 90 milliseconds to see what happens.

The second thing you must accomplish with this code is to advance to the next cell of animation. The value *tickDiff* is computed every time drawBubonic() is called. Listing 4-13 shows the code used to repeatedly draw the same animation cell for *n* milliseconds of time.

LISTING 4-13 Drawing the same cell for *n* milliseconds

```
DWORD thisTickCount;
DWORD tickDiff;

thisTickCount = GetTickCount();
tickDiff = thisTickCount - wartHog.lastTickCount;
```

If the value in *tickDiff* is greater than the value in *wartHog.delay* (90 milliseconds), you need to increment the animation cell ID, *wartHog.currentCell*, to index the next cell of animation. The value of *currentCell* is actually the index to the array standRightRect, used for blitting the source image to the back flip surface.



Case HOG_STOP_LEFT:

The complementary case for HOG_STOP_RIGHT is HOG_STOP_LEFT. This code looks identical to the previous example. The only difference is that you're going to have Bubonic facing the other direction. This is shown in Listing 4-14.

LISTING 4-14 Bubonic stops facing HOG_STOP_LEFT

```
case HOG_STOP_LEFT:
  drawOneToBack(wartHog.posx,wartHog.posy,
       standLeftRect[wartHog.currentCell], FALSE);

  if (tickDiff > wartHog.delay)
  {
    if (wartHog.currentCell++ > MAX_STOP_CELLS-1)
      wartHog.currentCell = 0;
    wartHog.lastTickCount = GetTickCount();
  }
  return;
```

The obvious difference is that the DDTOOLS function drawOneToBack() is using the *standLeftRect array*. You're using the cells of animation that have Bubonic standing facing to the left side of the display. Note that when you update the *currentCell* value, you are cycling through the values 0 to 3 (MAX_STOP_CELLS = 4). If you exceed the value MAX_STOP_CELLS, reset the *currentCell* index to 0.



Case HOG_GO_RIGHT:

The cases for HOG_GO_LEFT and HOG_GO_RIGHT are identical. The only difference is the RECT array used to identify the animation cells. The case for HOG_GO_RIGHT uses the array *goRightRect*, as opposed to HOG_GO_LEFT, which uses the array *goLeftRect*. With these considerations in mind, I'll only discuss the HOG_GO_RIGHT state, since the same code applies to the movement state HOG_GO_LEFT. This code is shown in Listing 4-15.

LISTING 4-15 Animation for Bubonic with state HOG_GO_RIGHT

```
case HOG_GO_RIGHT:
  drawOneToBack(wartHog.posx,wartHog.posy,
       goRightRect[wartHog.currentCell], FALSE);

  if (tickDiff > wartHog.delay)
  {
    if (wartHog.currentCell++ > MAX_WALK_CELLS-1)
      wartHog.currentCell = 0;
    wartHog.lastTickCount = GetTickCount();
  }
  return;
```

This case statement, like the HOG_STOP_RIGHT case statement, carries out two things. One, it blits the current animation cell from the offscreen surface to the back flip surface. Second, it checks to see if the delay for the current frame has expired. If it has, then the *currectCell* index is incremented or reset, depending on its value. The indices have a range of 0 to 4 (MAX_WALK_CELLS = 5).

Composing the Display

Putting it all together for simple animation, use the function updateFrame(). The code for updateFrame() is shown in Listing 4-16.

LISTING 4-16 Drawing the DIDRVR display

```
void updateFrame()
{
  EraseBackSurface();
  drawTerrain();
  drawBubonic();
  flipSurface();
}
```

The function to compose frames includes the DDTOOLS functions EraseBackSurface() and flipSurface(). Sandwiched between these calls are the two routines you developed for the DIDRVR application to animate Bubonic and give him a surface to run back and forth on.

The updateFrame() routine is called from the DIDRVR application's WinMain function, found in source file DIGENX.CPP. The frames of Bubonic are updated from the message loop of WinMain. Listing 4-17 contains the code for the message loop handling.

LISTING 4-17 The Message loop for the DIDRVR application

```
while(1)
{
  if(PeekMessage(&msg, NULL,0,0,PM_NOREMOVE))
  {
    if(!GetMessage (&msg, NULL, 0, 0))
      return msg.wParam;
    TranslateMessage (&msg);
    DispatchMessage (&msg);
  }
  else
  {
    // do critical game stuff
    updateFrame();
  }
}
```

If there are no messages for WinMain to process or dispatch, the program processes the animation cells of Bubonic and displays them for the user to see. At the moment, Bubonic doesn't do anything spectacular because you haven't provided any form of input.



Windows 95 Game SDK User Input

The simplest form of game input for any personal computer is the keyboard. Unfortunately, not all the game consumers in the world own a Pentium with CD-ROM, SoundBlaster 16 card, and a Gravis joystick. I, for example, own a 486 system with a $19.95 Gravis gamepad. For the longest time I was strictly a keyboard gameplayer. It used to drive my friends nuts when we played tournament games of DOOM. They always selected the mouse. I, on the other hand, have been playing games on the computer for so long, only using the keyboard as input, that I could beat the pants off of the best of them.



Keyboard Input

Computer systems are still expensive, and you can't expect a game consumer to own a joystick or control pad. It's likely that the game player owns at least a mouse. I can't see anybody trying to navigate Windows 95 strictly on keyboard input. However, start with the most common, and required, denominator, the PC keyboard. As you examine Windows 95 keyboard processing techniques, you'll add functionality to the DIDRVR application to provide movements for Bubonic the WartHog.



Keydown Message Event

The simplest form of input that Windows 95 has to offer is the WM_KEYDOWN message. You used this event back in Chapter 1 for processing the Esc key event. The Esc key is used to gracefully terminate the generic game application. Expanding upon the code you already wrote in Chapter 1, the new CALLBACK function WndFunc() is shown in listing 4-18.

LISTING 4-18 Using Windows messaging to capture keyboard input

```
case WM_KEYDOWN:
  switch(wParam)
  {
    case VK_NUMPAD4:
      wartHog.posx -= 8;
      if (wartHog.posx < (LEFT_BOUNDS))
      {
        wartHog.posx = LEFT_BOUNDS;
        wartHog.direction = HOG_STOP_LEFT;
      }
```

continued on next page

continued from previous page

```
      else
        wartHog.direction = HOG_GO_LEFT;
      return(0);
    case VK_NUMPAD5:
      switch(wartHog.direction)
      {
          case HOG_GO_RIGHT:
            wartHog.direction = HOG_STOP_RIGHT;
            break;
          case HOG_GO_LEFT:
            wartHog.direction = HOG_STOP_LEFT;
            break;
          default:
            break;
      } // switch direction
      return(0);
    case VK_NUMPAD6:
      wartHog.posx += 8;
      if (wartHog.posx > (RIGHT_BOUNDS))
      {
        wartHog.posx = RIGHT_BOUNDS;
        wartHog.direction = HOG_STOP_RIGHT;
      }
      else
        wartHog.direction = HOG_GO_RIGHT;
      return(0);
    case VK_ESCAPE:
      PostQuitMessage(0);
      break;
}
break;
```

You have added the keys VK_NUMPAD4 to go left, VK_NUMPAD5 to stop Bubonic, and VK_NUMPAD6 to go right. The VK in Windows lingo stands for virtual key. A virtual key represents a single stroke of a key on your keyboard. Just as there are scan codes defined for the keyboard, Windows has virtual keys defined as well. You can find them in your windows.h header file where they are all defined.



Moving Bubonic Right on WM_KEYDOWN

In this example, you're going to move Bubonic to the right using the key VK_NUMPAD6. To use the numeric keypad, the NumLock must be set. If you have problems with the demo program, check your NumLock key. The movement will be slow because you are relying on the Windows 95 operating system to post a message to your application each time it detects the VK_NUMPAD6 key is down.

In your application, when you detect the key is down, you need to update values in the wartHog structure to respond to the user's input. Listing 4-19 shows how Bubonic is updated to move to the right when the VK_NUMPAD6 key is pressed.

LISTING 4-19 Handling the WM_KEYDOWN event for VK_NUMPAD6

```
case VK_NUMPAD6:
  wartHog.posx += 8;
  if (wartHog.posx > (RIGHT_BOUNDS))
  {
    wartHog.posx = RIGHT_BOUNDS;
    wartHog.direction = HOG_STOP_RIGHT;
  }
  else
    wartHog.direction = HOG_GO_RIGHT;
```

Every time your DIDRVR application processes an event for the VK_NUMPAD6 key down, you need to update the wartHog structure. For the test application, Bubonic moves 8 pixels to the right with each VK_NUMPAD6 key down event. Bubonic has a limitation on how far he can travel over to the right of the display. The display is 640 pixels wide. Set a limit to the right of the screen using the equation 640 - 64 pixels (RIGHT_BOUNDS). If Bubonic's wartHog field *posx* ever contains a value greater than this, post the direction state HOG_STOP_RIGHT and the *posx* field is set to RIGHT_BOUNDS. If Bubonic is still within the bounds, terminate the key processing by setting the direct state to HOG_GO_RIGHT.

The code used to move Bubonic to the left is nearly identical. The primary differences are that you move in the -8 pixels direction, and set the left boundary to 32 (LEFT_BOUNDS). If Bubonic steps into the boundary on the left, the direction state is posted for HOG_STOP_LEFT. The *posx* field is then set to LEFT_BOUNDS.



Stopping Bubonic on WM_KEYDOWN

When the DIDRVR application processes the event for VK_NUMPAD5 key down, Bubonic is stopped. There are two direction states for stopping Bubonic: one, for if he is traveling to the right, and two, if he was heading left. Listing 4-20 shows how you stop Bubonic when the VK_NUMPAD5 key is pressed.

LISTING 4-20 Bubonic stops on WM_KEYDOWN message VK_NUMPAD5

```
case VK_NUMPAD5:
  switch(wartHog.direction)
```

continued on next page

continued from previous page

```
  {
    case HOG_GO_RIGHT:
      wartHog.direction = HOG_STOP_RIGHT;
      break;
    case HOG_GO_LEFT:
      wartHog.direction = HOG_STOP_LEFT;
      break;
    default:
      break;
} // switch direction
return(0);
```

Basically, all you do is check which way Bubonic is heading and stop him correspondingly. Note that you don't set any values of the wartHog field here. The *posx* value is only updated with each VK_NUMPAD4 and VK_NUMPAD6 key down event. Bubonic doesn't have any velocity values. This is because you only want him to respond to user inputs.



Getting Keyboard Input on Demand!

The Win32 SDK provides a function, GetAsyncKeyState(), that checks to see if a key is up or down at the time the function is called. The return value from GetAsyncKeyState() also tells you if the key has been pressed since the last time you called the function. When the most significant bit of the return value is set, the key is down. If the least significant bit is set, the key has been down since the last time the function was called. A return value of 0 indicates another application is currently using the keyboard.

To use the GetAsyncKeyState() function, you have to clean out the WM_KEYDOWN message handling routines for VK_NUMPAD keys you used previously. In fact, your new function, checkKeyboard(), won't even be called from the message loop. Listing 4-21 shows how keyboard states can be checked using GetAsyncKeyState.

LISTING 4-21 Checking the keyboard state on demand

```
void checkKeyboard()
{
  if (GetAsyncKeyState(VK_NUMPAD4)
  {
    wartHog.posx -= 8;
    if (wartHog.posx < (LEFT_BOUNDS))
    {
      wartHog.posx = LEFT_BOUNDS;
      wartHog.direction = HOG_STOP_LEFT;
    }
    else
```

```
      wartHog.direction = HOG_GO_LEFT;
  }

  if (GetAsyncKeyState(VK_NUMPAD5)
  {
    switch(wartHog.direction)
    {
      case HOG_GO_RIGHT:
        wartHog.direction = HOG_STOP_RIGHT;
        break;
      case HOG_GO_LEFT:
        wartHog.direction = HOG_STOP_LEFT;
        break;
      default:
        break;
    } // switch direction
  }
  if (GetAsyncKeyState(VK_NUMPAD6)
  {
      wartHog.posx += 8;
      if (wartHog.posx > (RIGHT_BOUNDS))
      {
        wartHog.posx = RIGHT_BOUNDS;
        wartHog.direction = HOG_STOP_RIGHT;
      }
      else
        wartHog.direction = HOG_GO_RIGHT;
  }
}
```

Your new routine, checkKeyboard(), is going to be called from updateFrame(). To speed up the keyboard response time, you want to retrieve keystrokes while the game is in its critical processing time and not during message dispatch and reading. The modifications to updateFrame() are shown in Listing 4-22.

LISTING 4-22 Using checkKeyboard() from within UpdateFrame()

```
void updateFrame()
{
  checkKeyboard();
  EraseBackSurface();
  drawTerrain();
  drawBubonic();
  flipSurface();
}
```

The function GetAsyncKeyState() is going to be utilized by commercial games over the keyboard event processing. Your demo only moves Bubonic eight pixels in either direction of the screen. Even at this pace, the significance of using GetAsyncKeyState() over keyboard messaging is apparent. This Win32

function handles the job of getting keystrokes for game applications rather efficiently. You can use the standard Windows keyboard message events for application notifications such as toggling audio, pausing a game, or terminating the application.



Mouse Input

The most common input device for the Windows 95 operating system is the mouse. For games like *Virtua-Fighter* or *Mortal Kombat*, the mouse isn't the greatest device to use for user command input. However, for tile games, role-playing games (RPGs), Tetris-style games, or Bubonic-the-WartHog-style games, the mouse is a very effective device for user input. In fact, its response in messaging is superior to that of keyboard input using WM_KEYDOWN.



Moving Bubonic with Mouse Input Messages

DOS game programmers are too used to working with encodings, called mickeys, to determine the mouse position on the screen. The Windows 95 operating system is a graphical interface, so logically all mouse coordinate information would be in terms of screen coordinates. To track the mouse movement and determine its direction of movement, you will need to create a special structure in your application. Listing 4-23 shows the declaration of the DIDRVR application's mouse structure.

LISTING 4-23 Mouse structure used by the DIDRVR application

```
typedef struct
{
  int posx;
  int posy;
  int oldposx;
  int oldposy;
  BOOL moved;
}MOUSE, *LPMOUSE;

MOUSE Mouse;
```

The windows message WM_MOUSEMOVE is used to handle a message from the operating system when the mouse has moved. When the DIDRVR application handles a mouse message, it updates the mouse structure with the current position of the cursor. To process mouse messages, you need to modify your callback WndFunc() function in source file DIGENX.CPP. Listing 4-24 shows the handling of the WM_MOUSEMOVE messages.

LISTING 4-24 Processing the WM_MOUSEMOVE message

```
case WM_MOUSEMOVE:
  if (Mouse.moved != TRUE)
  {
    Mouse.posx = LOWORD(lParam);
    Mouse.posy = HIWORD(lParam);
    Mouse.moved = TRUE;
  }
  return 0;
```

The message WM_MOUSEMOVE is a 32-bit integer. The mouse cursor coordinates are stored in the lParam field of the message. The Windows macros LOWORD and HIWORD are used to extract the coordinates from the message.

You can also add a message handler for the left mouse button to stop Bubonic, just as you used the VK_NUMPAD5 key in the keyboard input example. For your application to process mouse input messages, you need to add case statement WM_LBUTTONDOWN. This will handle the event when the user presses the left mouse button down. If you were to detect the right mouse button being pressed, you would use the case statement WM_RBUTTONDOWN. In this example, you will only use the left mouse button, shown in Listing 4-25.

LISTING 4-25 Handling the mouse WM_LBUTTONDOWN message

```
case WM_LBUTTONDOWN:
  switch(wartHog.direction)
  {
    case HOG_GO_RIGHT:
      wartHog.direction = HOG_STOP_RIGHT;
      break;
    case HOG_GO_LEFT:
      wartHog.direction = HOG_STOP_LEFT;
      break;
    default:
      break;
  } // switch direction
  return 0;
```

The code used to stop Bubonic, when the user presses the left mouse button down, is identical to your message handler for when the VK_NUMPAD5 key is down. If the user presses the left mouse button down, the *wartHog* direction state will be set to stop Bubonic from moving.

If the mouse button isn't down, then you need to process the coordinates being stored in the mouse structure with each WM_MOUSEMOVE message. The function you'll add to your DIDRVR application is handleMouse(). The ode for processing mouse movements is shown in Listing 4-26.

LISTING 4-26 Processing the mouse updates from handleMouse()

```
#define MIN_MOUSE_MOVE 3
#define SCREEN_CENTER_X 320
#define SCREEN_CENTER_Y 240

void handleMouse()
{
  int deltax;

  deltax = Mouse.posx - Mouse.oldposx;
  // do mouse update
  // moving in positive x direction (right)
  if (deltax > MIN_MOUSE_MOVE)
  {
    wartHog.posx += 8;
    if (wartHog.posx > (RIGHT_BOUNDS))
    {
      wartHog.posx = RIGHT_BOUNDS;
      wartHog.direction = HOG_STOP_RIGHT;
    }
    else
      wartHog.direction = HOG_GO_RIGHT;
  }

  if (deltax < -MIN_MOUSE_MOVE)
  {
    wartHog.posx -= 8;
    if (wartHog.posx < (LEFT_BOUNDS))
    {
      wartHog.posx = LEFT_BOUNDS;
      wartHog.direction = HOG_STOP_LEFT;
    }
    else
      wartHog.direction = HOG_GO_LEFT;
  }

  Mouse.moved = FALSE;
  Mouse.oldposx = SCREEN_CENTER _X;
  Mouse.oldposy = SCREEN_CENTER _Y;
  SetCursorPos(SCREEN_CENTER _X, SCREEN_CENTER _Y);
}
```

Every time the mouse moves, you read the mouse cursor position. After processing the coordinates, you call the Windows API routine SetCursorPos() to move the cursor to the center of the screen with coordinates SCREEN_CENTER_X and SCREEN_CENTER_Y. Figure 4-12 shows the user moving the mouse to the right. These coordinates are stored in the mouse structure *oldposx* and *oldposy*. When you process the mouse coordinates, you calculate a *deltax* value. The distance delta tells you how far to the right (*deltax* > 0) or how far to the left (*deltax* < 0) the user has moved the mouse. In this example, the show cursor flag is set to TRUE. You watch the distance the mouse travels from the center of the screen with each mouse message.

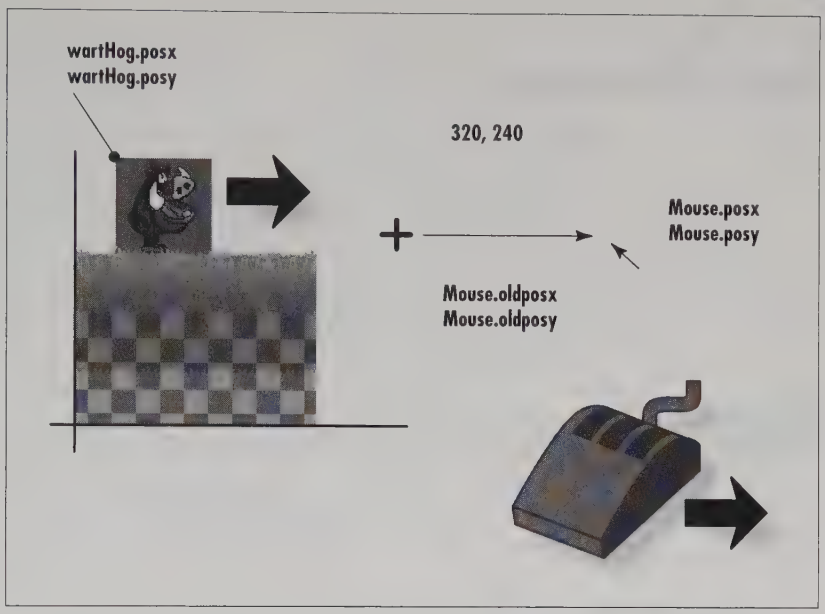


FIGURE 4-12
Moving the mouse to the right

When you handle the mouse movement, you read the mouse cursor position. Using the position coordinates, calculate the distance the cursor has moved from the middle of the screen. If the *deltax* is greater than the MIN_MOUSE_MOVE threshold, move Bubonic to the right or the left. MIN_MOUSE_MOVE is a fine-tuning value. The slightest vibration can move a mouse, so check to see if the mouse has actually been moved, or if it was merely jiggled on the pad.

The mouse-handling routine is called from updateFrame() in source file DIGENX.CPP. With the keyboard drivers already present, updateFrame() is shown in Listing 4-27.

LISTING 4-27 Using handleMouse() from updateFrame()

```
void updateFrame()
{
  checkKeyboard();
  handleMouse();
  EraseBackSurface();
  drawTerrain();
  drawBubonic();
  flipSurface();
}
```

Unlike keyboard input messages, mouse input messaging works rather well with the Microsoft Direct X Game SDK. Cases in which gameplay becomes intensive, such as with *Doom95*, the mouse input messaging will be inadequate.



Mouse on Demand

Some graphics-intensive programs may require *...more power. Arrr! Arrr! Arrr!*

You can bypass the mouse messaging and retrieve the mouse cursor position and mouse buttons on demand. The Windows API call GetCursorPos() returns the values of the current cursor position. To use the Windows API call, you need to remove the WM_MOUSEMOVE and WM_LBUTTONDOWN message cases from the callback WndFunc() function in DIGENX.CPP. Next you need a new function to internally process mouse events. Listing 4-28 shows the code for getMouse().

LISTING 4-28 Getting mouse updates on demand

```
void getMouse()
{
  POINT pt;
  int deltax;

  if (!GetCursorPos(&pt) )
    return;

  Mouse.posx = pt.x;
  Mouse.posy = pt.y;

  deltax = Mouse.posx - Mouse.oldposx;
  // do mouse update
  // moving in positive x direction (right)
  if (deltax > MIN_MOUSE_MOVE)
  {
    wartHog.posx += 8;
    if (wartHog.posx > (RIGHT_BOUNDS))
    {
      wartHog.posx = RIGHT_BOUNDS;
      wartHog.direction = HOG_STOP_RIGHT;
    }
    else
      wartHog.direction = HOG_GO_RIGHT;
  }

  if (deltax < -MIN_MOUSE_MOVE)
  {
    wartHog.posx -= 8;
    if (wartHog.posx < (LEFT_BOUNDS))
    {
      wartHog.posx = LEFT_BOUNDS;
      wartHog.direction = HOG_STOP_LEFT;
    }
    else
      wartHog.direction = HOG_GO_LEFT;
```

```
  }

  if (GetAsyncKeyState(VK_LBUTTON))
  {
    switch(wartHog.direction)
    {
      case HOG_GO_RIGHT:
        wartHog.direction = HOG_STOP_RIGHT;
        break;
      case HOG_GO_LEFT:
        wartHog.direction = HOG_STOP_LEFT;
        break;
      default:
        break;
    } // switch direction
  }

  Mouse.moved = FALSE;
  Mouse.oldposx = CENTER_X;//Mouse.posx;
  Mouse.oldposy = CENTER_Y; //Mouse.posy;
  SetCursorPos(CENTER_X, CENTER_Y);
}
```

The function getMouse() is nearly identical to the mouse message input handler handleMouse(). The significant change is that you added the Windows POINT variable, *pt*, to use with the Windows API call GetCursorPos(). The information from the POINT structure *pt* is then moved into the Mouse struct for the *deltax* calculation. As you can see, the code is identical to your previous routine handleMouse().

A significant change to the body of code is the left mouse button handler. In the getMouse() function, you see that you can use the Win32 call GetAsyncKeyState(). This is the function you used to bypass WM_KEYDOWN messages for keyboard input. This routine is also useful in retrieving the mouse buttons. The code to do this is shown in Listing 4-29.

LISTING 4-29 Using GetAsyncKeyState() to capture mouse button input

```
if (GetAsyncKeyState(VK_LBUTTON))
{
  switch(wartHog.direction)
  {
    case HOG_GO_RIGHT:
      wartHog.direction = HOG_STOP_RIGHT;
      break;
    case HOG_GO_LEFT:
      wartHog.direction = HOG_STOP_LEFT;
      break;
    default:
      break;
  } // switch direction
}
```

As before, this code is identical to how you handled the VK_NUMPAD5 key with GetAsyncKeyState(). If you want, you can add the right mouse button to stop Bubonic.

Using your routine getMouse() works sufficiently well for demanding applications but is still too slow for intensive applications. Microsoft is aware of the mouse input short-comings and is reportedly working on a DirectMouse API for the DirectX API suite of game development tools.



DirectInput

Windows 95 ships with a component of DirectInput already installed. This library is actually found in the Win32 SDK module WINMM.LIB rather than a DirectX library. DirectInput, like its DirectX API counterparts, is a device-independent interface to joysticks, game flight simulator controllers, and other forms of digital and analog game input. The criteria for using DirectInput is that the users have a device driver for their hardware that Windows 95 can recognize. Control devices that use DirectInput must be calibrated in the Windows 95 control panel utility for joysticks.



Moving Bubonic with DirectInput

You're going to add code to your input test application, DIDRVR, so you can use the DirectInput API included with the Win32 SDK. To use a device with DirectInput, it must first be calibrated by the Windows 95 control panel utility joystick. Second, the application utilizing the DirectInput-compatible device must initialize joystick data structures. The Win32 library originally supported a message-based joystick handler with a JOYINFO data structure. This method of handling joysticks has been permanently archived to the bit bucket. Although the Win32 library still supports this form of input, the new structure for DirectInput is JOYINFOEX. Figure 4-13 shows the new Win32 structure that is used specifically by Windows 95. This new architecture allows 16 joysticks, each with six axes of movement, and two active buttons each, to communicate on one PC simultaneously. Analog joysticks are more hardware intensive, and only four of these are supported with two axes of movement.

To initialize your DirectInput joystick driver for the DIDRVR application, you are going to need to provide a data structure of your own to maintain updates. The custom structure, JOYDATA, is shown in Listing 4-30.

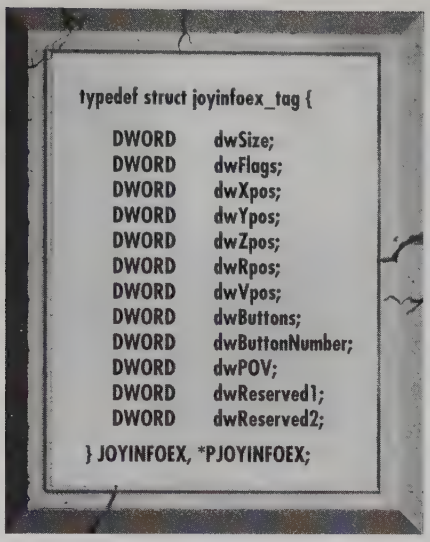


FIGURE 4-13
The winmm structure JOYINFOEX

LISTING 4-30 The structure JOYDATA used for DirectInput

```
typedef struct JOYDATA_tag
{
  UINT        JoyId;
  JOYCAPS     JoyCaps;
  JOYINFOEX   JoyInfoEx;
  JOYINFOEX   JoyOldInfoEx;
} JOYDATA, *LPJOYDATA;

LPJOYDATA pJoyData;
```

This structure contains information about your input device capabilities and stores information about the current and previous positions of the joystick in use. Your first step in adding DirectInput to your input test application is to initialize your JOYDATA structure as shown in Listing 4-31.

LISTING 4-31 Initializing DirectInput

```
BOOL joystick = FALSE;

BOOL initJoyStick()
{
  MMRESULT mmres;
  joystick = FALSE;

  if (joyGetNumDevs())
  {
    joystick = TRUE;
```

continued on next page

continued from previous page

```
    pJoyData = (LPJOYDATA)malloc(sizeof(*pJoyData));
    if (!pJoyData)
      return FALSE;
    pJoyData->JoyId = 0;
    mmres = joyGetDevCaps(0,&pJoyData->JoyCaps, sizeof(pJoyData->JoyCaps));
    if (mmres == JOYERR_NOERROR)
    {
      memset (&pJoyData->JoyInfoEx, 0, sizeof(pJoyData->JoyInfoEx));
      pJoyData->JoyInfoEx.dwSize = sizeof(pJoyData->JoyInfoEx);
      pJoyData->JoyInfoEx.dwFlags = JOY_RETURNBUTTONS |
                  JOY_RETURNX | JOY_RETURNY | JOY_USEDEADZONE;
      if( pJoyData->JoyCaps.wCaps & JOYCAPS_HASZ )
        pJoyData->JoyInfoEx.dwFlags |= JOY_RETURNZ;
      if( pJoyData->JoyCaps.wCaps & JOYCAPS_HASR )
        pJoyData->JoyInfoEx.dwFlags |= JOY_RETURNR;
      if( pJoyData->JoyCaps.wCaps & JOYCAPS_HASU )
        pJoyData->JoyInfoEx.dwFlags |= JOY_RETURNU;
      if( pJoyData->JoyCaps.wCaps & JOYCAPS_HASV )
        pJoyData->JoyInfoEx.dwFlags |= JOY_RETURNV;
      if( pJoyData->JoyCaps.wCaps & JOYCAPS_HASPOV )
        pJoyData->JoyInfoEx.dwFlags |= JOY_RETURNPOV;
    } // mmres == NOERR_JOYERR
  } // joyGetNumDevs()
  return (BOOL)joystick;
}
```

Using the Direct API call joyGetNumDevs(), you query the joystick driver for the number of joysticks that it can support. If the call is successful, it will return an unsigned integer greater than 0. This value is the number of input devices the DirectInput device driver can support.

If your device driver query returns a non-zero integer, you can initialize your *pJoyData* structure. You have previously declared your variable *pJoyData* as a pointer to the JOYDATA structure. You need to allocate memory for the struct and begin initializing the fields of the structure. This code is shown in Listing 4-32.

LISTING 4-32 Allocating and initializing the pJoyData structure

```
HRESULT mmres;

pJoyData = (LPJOYDATA)malloc(sizeof(*pJoyData));
if (!pJoyData)
  return FALSE;
joystick = TRUE;
pJoyData->JoyId = 0;
mmres = joyGetDevCaps(0,&pJoyData->JoyCaps,
                      sizeof(pJoyData->JoyCaps));
```

The BOOL variable joystick is a flag used by the application to indicate that the joystick has been initialized. The field *JoyId* is used to identify the current joystick. This example only uses one game controller. Had there been several other controllers attached, you would use an array of JOYDATA struct to handle the multiple-input devices. The *JoyId* field would identify the current input device being processed.

With your JOYDATA struct allocated, you need to query the input device of its capabilities. The DirectInput API call joyGetDevCaps() performs this operation. Figure 4-14 shows the prototype of this function. Your JOYDATA struct contains a DirectInput struct, JOYCAPS. This struct needs to be passed to joyGetDevCaps() to initialize your own JOYDATA struct.

Once the JOYDATA's field struct JOYCAPS has been filled in, you can start setting the flags to the DirectInput JOYINFOEX struct in JOYDATA. This code is shown in Listing 4-33.

LISTING 4-33 Setting the JOYINFOEX and JOYCAPS flags

```
if (mmres == JOYERR_NOERROR)
{
  memset (&pJoyData->JoyInfoEx, 0, sizeof(pJoyData->JoyInfoEx));
  pJoyData->JoyInfoEx.dwSize = sizeof(pJoyData->JoyInfoEx);
  pJoyData->JoyInfoEx.dwFlags = JOY_RETURNBUTTONS |
                    JOY_RETURNX | JOY_RETURNY | JOY_USEDEADZONE;
  if( pJoyData->JoyCaps.wCaps & JOYCAPS_HASZ )
    pJoyData->JoyInfoEx.dwFlags |= JOY_RETURNZ;
  if( pJoyData->JoyCaps.wCaps & JOYCAPS_HASR )
    pJoyData->JoyInfoEx.dwFlags |= JOY_RETURNR;
  if( pJoyData->JoyCaps.wCaps & JOYCAPS_HASU )
    pJoyData->JoyInfoEx.dwFlags |= JOY_RETURNU;
  if( pJoyData->JoyCaps.wCaps & JOYCAPS_HASV )
    pJoyData->JoyInfoEx.dwFlags |= JOY_RETURNV;
  if( pJoyData->JoyCaps.wCaps & JOYCAPS_HASPOV )
    pJoyData->JoyInfoEx.dwFlags |= JOY_RETURNPOV;
} // mmres == NOERR_JOYERR
```

In this body of code, you are setting the flags for the desired information you want returned from the DirectInput JOYINFOEX struct. Since this is a test application, you informed DirectInput to return practically all information it finds. You won't use all of this information in your current DirectInput test application. However, later, when you're approached by your manager to try out a new device, you're more than halfway started into prototyping a DirectInput test application for the new device.

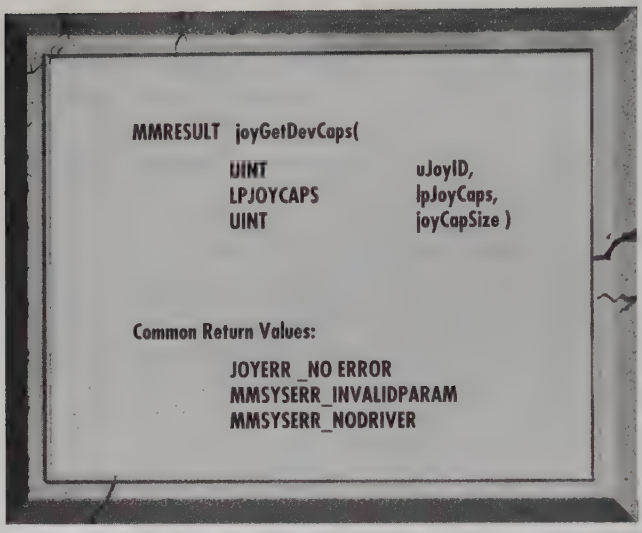


FIGURE 4-14
Getting the device joystick capabilities

Processing Device Input with DirectInput

You have your function, initJoyStick(), that initializes practically everything supported by DirectInput. Now you need a function to process the joystick input and move Bubonic around on the screen. Listing 4-34 provides the code for the getJoyStick() function.

LISTING 4-34 The getJoystick() function

```
void getJoystick( void )
{
  MMRESULT   mmres;
  int        joyPosX;

  if (!joystick)
    return;
  pJoyData->JoyOldInfoEx = pJoyData->JoyInfoEx;
  mmres = joyGetPosEx (pJoyData->JoyId, &pJoyData->JoyInfoEx);
  if (mmres != JOYERR_NOERROR)
    return;
  joyPosX = normalize(pJoyData->JoyInfoEx.dwXpos,
                      pJoyData->JoyCaps.wXmin,
                      pJoyData->JoyCaps.wXmax,640);
  if (joyPosX < 280)
  {
    wartHog.posx -= 8;
```

```
    if (wartHog.posx < (LEFT_BOUNDS))
    {
      wartHog.posx = LEFT_BOUNDS;
      wartHog.direction = HOG_STOP_LEFT;
    }
    else
      wartHog.direction = HOG_GO_LEFT;
  }
  if (joyPosX > 360)
  {
    wartHog.posx += 8;
    if (wartHog.posx > (RIGHT_BOUNDS))
    {
      wartHog.posx = RIGHT_BOUNDS;
      wartHog.direction = HOG_STOP_RIGHT;
    }
    else
      wartHog.direction = HOG_GO_RIGHT;
  }

  // Button one has changed state.
  if ((pJoyData->JoyInfoEx.dwButtons
       ^ pJoyData->JoyOldInfoEx.dwButtons) & BUTTON1)
  {
    if (pJoyData->JoyInfoEx.dwButtons & BUTTON1)
    {
      switch(wartHog.direction)
      {
        case HOG_GO_RIGHT:
          wartHog.direction = HOG_STOP_RIGHT;
          break;
        case HOG_GO_LEFT:
          wartHog.direction = HOG_STOP_LEFT;
          break;
        default:
          break;
      } // switch direction
            }
    } // if BUTTON1
  }  // BUTTON changed state
}
```

The code looks similar to getMouse(). This is actually a cut-and-paste rendition of the mouse routine modified to use DirectInput. Instead of calling GetCursorPos(), as you did with the mouse function, you call the DirectInput routine, joyGetPosEx(). Figure 4-15 shows the details of this DirectInput API call.

When the API routine joyGetPosEx() is called, you pass as calling arguments the joystick ID from your JOYDATA structure and the DirectInput JOYINFOEX struct also stored in JOYDATA. Listing 4-35 shows how you check the joystick input.

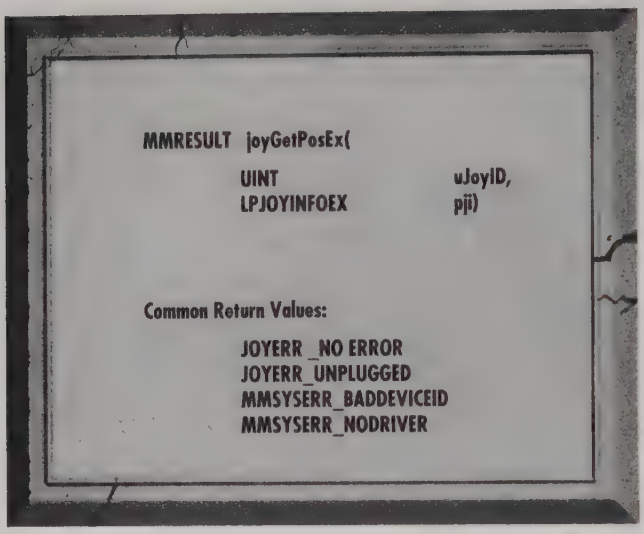


FIGURE 4-15
Getting the joystick position

LISTING 4-35 Retrieving and normalizing the joystick coordinates

```
MMRESULT      mmres;
int           joyPosX;

if (!joystick)
  return;
pJoyData->JoyOldInfoEx = pJoyData->JoyInfoEx;
mmres = joyGetPosEx (pJoyData->JoyId, &pJoyData->JoyInfoEx);
if (mmres != JOYERR_NOERROR)
  return;
joyPosX = normalize(pJoyData->JoyInfoEx.dwXpos,
            pJoyData->JoyCaps.wXmin,
            pJoyData->JoyCaps.wXmax,640);
```

In this example you are only interested in two axes of motion, *x* and *y*. Actually, you only respond to the horizontal axes of motion, but the vertical values are currently being returned as well. To determine the current *x* position of your joystick in your logical playing field, you need to normalize the coordinate value. The code for normalizing the joystick input values is shown in Listing 4-36.

LISTING 4-36 Normalizing an integer value

```
int normalize (int val, int minval, int maxval, int range)
{
  // error detection:
  if ((maxval-minval) == 0)
  {
    return(0);
  }
```

```
  // zero-base:
  val -= minval;

  // normalize to range
  val = (range * val) / (maxval-minval);

  return(val);
}
```

The values you pass to normalize() are, the *pJoyData->JoyInfoEx.dwXpos*, which is the current position of the joystick (*val*), and the min and max *x* coordinate values of the joystick found in the JOYDATA structure *JoyCaps* field. The range you supply for the *joyPosX* normalization process is the width of the playing screen, which is 640 (range) pixels. When normalization of the value is completed, you have a value that's suitable to your coordinate space. With this value, you can move Bubonic in response to the user's input. The code in Listing 4-37 demonstrates how Bubonic moves in response to joystick input.

LISTING 4-37 Moving Bubonic with DirectInput

```
// go left
if (joyPosX < 280)
{
  wartHog.posx -= 8;
  if (wartHog.posx < (LEFT_BOUNDS))
  {
    wartHog.posx = LEFT_BOUNDS;
    wartHog.direction = HOG_STOP_LEFT;
  }
  else
    wartHog.direction = HOG_GO_LEFT;
}

// go right
if (joyPosX > 360)
{
  wartHog.posx += 8;
  if (wartHog.posx > (RIGHT_BOUNDS))
  {
    wartHog.posx = RIGHT_BOUNDS;
    wartHog.direction = HOG_STOP_RIGHT;
  }
  else
   wartHog.direction = HOG_GO_RIGHT;
}
```

The values of 280 and 360 used above are calibrations of 320 + 40 pixels. These are deltas from the center of the screen that produced the best results with the device used in the test. You may want to experiment with these values to see what works best with your input device. Other than the calibration values, this is a re-hack of the mouse code.



Stopping Bubonic with a Joystick Button

The last piece of code to present is the driver for the joystick buttons. In your source code, you define the global BUTTON1 to identify your joystick button number one. Every time the input device is polled by the application, the button states are stored in the DirectInput JOYINFOEX struct, in the field dwButtons. What you want to do is examine the state change of the buttons to see if the user pressed BUTTON1 to stop Bubonic. This is shown in Listing 4-38.

LISTING 4-38 Stop Bubonic using joystick button one

```
#define BUTTON1 1
// Button one has changed state.
if ((pJoyData->JoyInfoEx.dwButtons ^
  pJoyData->JoyOldInfoEx.dwButtons) & BUTTON1)
{
  if (pJoyData->JoyInfoEx.dwButtons & BUTTON1)
  {
    switch(wartHog.direction)
    {
      case HOG_GO_RIGHT:
        wartHog.direction = HOG_STOP_RIGHT;
        break;
      case HOG_GO_LEFT:
        wartHog.direction = HOG_STOP_LEFT;
        break;
     default:
       break;
   } // switch direction
      }
} // if BUTTON1
```

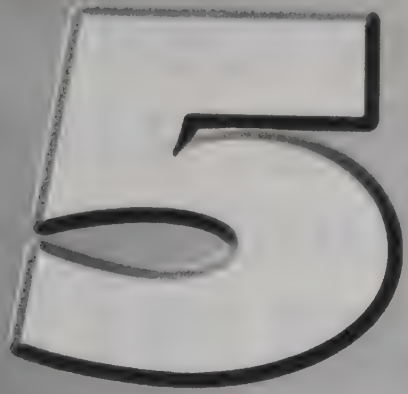
The JOYDATA struct that was provided to store JOYINFOEX data contains a current instance of input data in *JoyInfoEx*, and a previous instance in *JoyOldInfoEx*. Using logical operators on these values will tell you if the device button state has changed. If the *pJoyData->JoyInfoEx.dwButtons* is 0, then no button has been pressed. If it contains a 1, then BUTTON1 has been pressed and so on. You are only interested in BUTTON1 for your test. If BUTTON1 is pressed, set the direction states for the *wartHog* structure. This code you have seen on several occasions for both keyboard and mouse input.





Interfacing the Artifact

Out of all the DirectX APIs, DirectInput is going to be one of the most fun for developers and hobbyists to tinker with. This interface should have been called the DirectToy API because of all the cool devices that are being designed to interface to the DirectInput API. You'll take a look at one such device in Chapter 12. And don't skip ahead. It's not fair to peek. Well, class has been in session long enough. I guess a dismissal is in order before you start to take a look at DirectSound in the next chapter.



DIRECTSOUND: ENTER THE REALM OF THE QUEEN OF THE NIGHT



DIRECTSOUND: ENTER THE REALM OF THE QUEEN OF THE NIGHT

I have been a video game addict and programmer for at least 15 years now. I am including the years when my game programming skills set was in its infancy. My abilities were nurtured on a TRS-80 Model I computer with 16 K of RAM. A company called Big Five Software, run by some inspiring young developers, produced the top game hits on 128-by-48 character game displays. *Big Five's* first big hit was a game called *SuperNova*. It was a port of *Asteroids* from vector graphics to block character graphics mode. It's amazing how many hours I spent playing this game. When this game finally fell to the wayside, *Big Five* released the block mode version of *Defender*, called *Eliminator*. The key selling point to *Eliminator* was that it had sounds. Actually, they were best described as audible tweaks. The game shipped on audio cassette, as was the mass storage norm back in 1980. Sound produced by the game was played through the speaker on the cassette deck player that was used to load the game. I remember being amazed at how the Big Five developers produced these recognizable tweaks of explosions and even some digitized voice through the simple speaker of the cassette player.

Computer sound technology has certainly come a long way in the last decade and a half. Like the cassette speaker port, many sound technologies have come and gone. In the late eighties, the Macintosh, the Apple IIGS, and the Amiga clearly had the nouveau sound systems. The PC had the AdLib sound board, which was the game industry standard for some time. Then came Creative Lab's SoundBlaster board. Its architecture became a superior presence in the industry during the early 90s, and continues to this day.

Working at Activision in the late eighties, I had the privilege of playing with all the sound card technologies firsthand. I got to work with sound cards I normally wouldn't have been able to afford. My own interests were focused on graphics card technology, so on my own personal budget, PC sound cards weren't given a high priority. At work I could pop in any new pricey piece of sound hardware and experiment with it. While at home, I pursued my insatiable need for gameplay in silence.

Two years ago, I made the transition to gaming hi-fidelity. It was the Electronic Arts game, *SpaceHulk*, that changed the way I viewed sound cards and their integration into a video game. This is the first game I played that truly utilized sound as a dimension to highlight the gameplay. Other games before *SpaceHulk* used sound and music to intensify gameplay. But *SpaceHulk* was a composition of sound and music that worked with the game players to make them feel as if they were a part of this make-believe world based on the successful RPG game. *SpaceHulk* was probably one of the first video games to have a monologue that was quotable and which people could recognize like a movie monologue. Anybody who has played the game has probably quoted the imperial leader with his line "Brothers...let the slaughter begin." The dark-sounding imperial leader sets the tone for the gameplay. Then the screen fades to start the game. You find yourself standing, with a team of your *SpaceHulk* marines, staring down a dark corridor. Ahead of you, screams shriek in the distance and become louder as the hideous gene stealers approach. Soon, there will be no escaping the shrieks as the intelligent gene stealers surround your heavily armed party. You will try to seek refuge and save what little of your party is left to make a last stand. But the shrieks become maddening and break your concentration. There is no way out. You can't go forward, because you can hear them ahead of you. You can't retreat back because it's too quiet. That's where they want you to go.

The dimension of sound increases the sensation of the game. Like *SpaceHulk*, sound is required for gameplay and is not just an attribute for a marketing selling point. I don't know how playable *SpaceHulk* is without sound. The player must rely on the 3D Doppler style sounds to determine enemy position when the sensors are damaged. Integrating sound rather than adding sound is a crucial element for successful fantasy games such as *SpaceHulk*, *DOOM*, and *Marathon* (on the Macintosh). Adding sound and music can set the tone of the game. Choreographing the sounds, though, can create a sense of awareness or *being* in the game world.

EPISODE THE MAGIC FLUTE



"Wherever you are, Bary, you are locked in," 13 commented.

"Well, I would rather be here than back there with those bloodsuckers. It's going to be a moment before I can catch my breath."

"Take your time, Bary. I'm trying to get a fix on your location. It's odd that even in virtual reality, physical activity such as running exerts a demand on your body."

"13, are you sure you're not half-Vulcan?" I joked as he tried to catch his breath. 13 was right. This is a trained response. How could I be out of breath if I haven't run anywhere nor have a body? I'm just a sprite.

I thought for a moment, and then decided to experiment. I wondered if I could be whatever I wanted. I concentrated further. Could I be whatever I wanted to be? When I has a kid, I had always wanted to be the mutant superhero, Captain Colossus. Well, it was worth a try.

"Armor on!" I yelled in my Captain Colossus universe-saving, feared-by-all-evil voice.

"Bary, what are you doing? I am reading a power surge from your PowerSuit."

I morphed to twice my virtual size and five times my build. Metallic alloy armor encased my virtual epidermis. It was the indestructible alloy that fused with Captain Colossus during an atomic bomb accident in the 1960s.

Kala entered the secret lab of 13 carrying her notes on the Agamemnon artifact. She stared at the giant display on the wall, where 13 monitored me in the artifact.

"Oh my God, what happened to Bary?"

"Nothing, this is just a case of *godly* powers placed in the hands of a juvenile."

Kala set her notes down on the chassis of 13's supercomputer.

"Bary, be careful of what you imagine next. We've got mixed company now," 13 reported.

"Oh, is Kala there? Hey, Kala, look at me. I'm Captain Colossus." I put my hands on his hips in the familiar superhero stance.

"Is Bary in the artifact?" Kala questioned.

"Yes. Why? Is something wrong, Kala?" 13 inquired.

"Something is very wrong. We are looking at architecture from the early 1800s, maybe even as early as 1780." Kala studied the display further.

"Can you pinpoint the century?" 13 asked.

"No. I studied two semesters of art history as a prerequisite in college to study ancient Egypt. This is something very recent. It's at the tip of my tongue. I know what this is." Kala studied further.

"Hey, guys, check out these strange little tube guys in the walls. They look like the magic flute from H.R. Puffenstuff."

"Bary, you hit it on the nose. This room you are in, Bary, isn't ancient Egyptian. It's a set from an 1800s opera, *The Magic Flute*, kind of an 1800s Spielburg production about Egyptian mythology. Bary, move forward into the next chamber. I remember this opera. We're in Act I, scene 15, *Entrance to the Realm of the Queen of the Night*."

"Say what? Kala, you make this sound like I'm trapped in the Carmen San Diego game from hell," Bary replied.

"Just go through the doors, Captain Clodhead," 13 joked.

"Beware evil-doers, Captain Colossus is here." With my increased virtual density, I knocked down a heavy sandstone door like it was cardboard.

"Way to go, Captain Clutz. You eliminated our element of surprise," 13 informs. Looking to Kala he asks, "Well, are you right?"

"13, there's one thing you'll learn about me. I'm always right. But...I don't recognize the object in the middle of the courtyard."

"Sorry, Captain Colossus. Can you take us over to the center of courtyard you're in. And please don't touch anything."

I strutted my armored, mutantly muscle-bulged body over to the object.

"13, this is weird. It's a statue of a sitting woman. Only I get the weird feeling it's alive. I feel like I've been dropped in an Anne Rice novel."

"You're not afraid of it, are you, Captain Colossus?" 13 joked.

"Bary is right. This statue is authentic ancient Egyptian. It doesn't fit the rest of the decor in this courtyard."

"This is too weird for Captain Colossus. Can the Justice League of the Universe move on now? This bogus music is getting to me."

"What music?" Kala asked.

13 piped it through to the speakers on his system.

"Bary, that bogus music is Mozart. He wrote the opera. If you ask me, we are in a burial ground. I think you should move on."

"Captain Colossus 10-4s that. Did anybody notice the tear on the statue being there earlier?"

"Negative, Captain Colossus. But I am powering you back down to PowerSuit mode. You're consuming far too much energy. We must conserve energy for your assault on the fissionable material storage and central computing core," 13 commanded.

"Oh, just a few more seconds; I didn't get to see what it feels like to fly," I pleaded.

I moved on out of the courtyard, but couldn't help but feel the statue was aware of my presence. It was an eerie feeling, like a soul was trapped in there.

But why did this soul reside in a garden of such beauty and splendor? Agamemnon controlled all this. He must have made this garden for somebody. Was it a loved one? Was this demon even capable of love?



DirectSound Audio File Formats

DirectSound is an interface for game programmers to access and query audio hardware running on a Windows 95 platform. DirectSound interfaces are hardware-independent. Like DirectDraw, DirectSound is a hardware abstraction. (See the discussion on HAL in chapter 3.) DirectSound is available to audio hardware that provides DirectSound device drivers. Manufacturers whose hardware support DirectSound include Creative Labs, nVidia, and Diamond, to name a few. This list is growing larger every day.

DirectSound provides a wider range of capabilities than the WAVE APIs included with the Microsoft multimedia suite of tools. WAVE API calls, such as PlaySound, are only useful for playing single audio streams. Another consideration is that the WAVE API and DirectSound cannot access audio hardware simultaneously. When the DirectSound API call SetCooperative() is used, all WAVE API calls will fail.

Although the WAVE APIs are not used in DirectSound, the WAVE format is still supported. DirectSound's focus is to provide a hardware abstraction through software interfaces. The DirectSound API doesn't provide any mechanisms for loading a WAVE file. This task is left to the programmer to read in a WAVE file and then present the sound sample to DirectSound as a Microsoft PCMWAVEFORMAT data structure.

The Microsoft Multimedia Development Kit (MDK) offers tools to load and unpack WAVE files. The MDK uses a specification called a RIFF file format to load sound samples into Windows 95.



RIFF Waveform Audio Files

The Resource Interchange File Format (RIFF) is the file structure used to store and retrieve the Microsoft Waveform Audio Format (WAV). The origin of the RIFF file organization stems from the Electronic Arts Interchange File Format (IFF) structure. Those of you who have spent countless hours developing your own Deluxe Paint LBM file unpackers will recognize this file format right off the bat. The IFF structure is a graphics file format more commonly known to developers as the IFF Interleaved Bitmap (ILBM) file format. During the mid-eighties the Commodore Amiga was the industry standard for computer artists creating art for PC games. The Electronic Arts program *Deluxe Paint II* stores its pictures and brushes in the same format on the IBM as it did on the Amiga. The

ILBM format is defined by Commodore, but PC programmers have used it for years, using the name LBM format. The LBM name stuck because it was the three-character file extension, ".LBM", used by EA to identify ILBM files on a PC.

The complete definition of an IFF file can be found in the Electronic Arts documentation, *EA IFF 85 Standard for Interchange File Format Files*. I obtained my copy of this article for a project I was working on about three years ago. Back then copies were scarce, and even EA had trouble locating the document. The primary objective for the IFF was to be a method for archival of structured graphics across different platforms. It was a format designed to tackle a lot of problems encountered by PCs when file sharing with other computers, such as the Amiga and the Macintosh. The IFF is a composite format, as you'll see in a moment, that provided the building blocks for storing graphics, animation sequences, and, later, sound. The term multimedia wasn't really in existence back in 1985. But the IFF was a strong and very successful effort to design a multimedia file format.



The Interchange Format File (IFF)

The idea behind the IFF file format is that all data is stored in tagged chunks. All the tagged data chunks are stored inside a large chunk called a form. All forms and chunks had specific properties to identify them. For instance, the first four bytes of the form and chunks contain labels identifying the data chunks. Right after the four-byte ID is an unsigned long value identifying how large the chunk is. For the form chunk, the size value identifies how big the file is, minus the four bytes for the chunk ID. In a data chunk, the chunk size value reflects how big the individual data chunk is, minus the four bytes for the chunk ID. An example of an IFF file is shown in Figure 5-1.

Programs written to read IFF files are typically called unpackers, meaning they unpack data chunks. Cutting and pasting from some old C code, the declaration for a form chunk is shown in Listing 5-1.

LISTING 5-1 Structure for the IFF form chunk

```
typedef struct
{
  unsigned long type;
  long  cksize;
  unsigned long subtype;
} formChunk;

formChunk iffFormChunk;

fread(&iffFormChunk,1,sizeof(formChunk), fp);
```

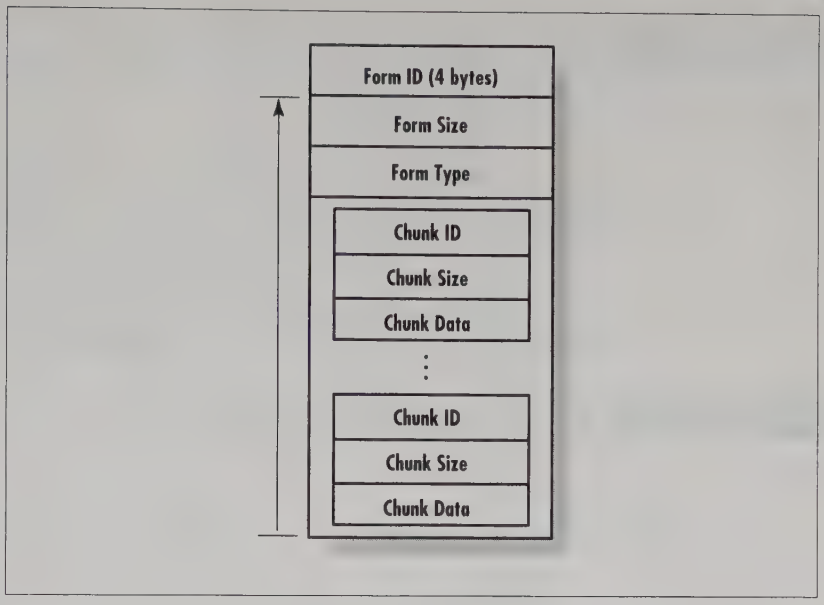


FIGURE 5-1
An IFF file with a form chunk and data chunks

This code would read in the form chunk preamble in the structure *iffFormChunk*. From there you could examine the contents of a *Deluxe Paint II* LBM file to see if it is an "ILBM" file. This could be done by examining the field *iffFormChunk.type*. If the file is identifiable by the unpacker, the code would then start an endless loop to unpack the data chunks. At this point, the file pointer is referencing a chunk data header, shown in Listing 5-2.

LISTING 5-2 Structure for the IFF chunk header

```
const unsigned int BYTE_LEN = 4;
typedef struct
{
  char      ckID[BYTE_LEN];
  unsigned long ckSize;
} ChunkHeader;

ChunkHeader chunk;
```

In the loop you execute another fread() to load the chunk header from file into the structure chunk. The four-byte field, *ckID*, of the chunk is then compared against a four-byte value you're searching to unpack. For the *Deluxe Paint II* files valid chunk header IDs are Bitmap Header, "BMHD"; LBM file image data, "BODY"; and the 256-color triplet palette, "CMAP". The declaration of the four-byte IDs is provided in Listing 5-3.

LISTING 5-3 Four-byte chunk ID declarations

```
const unsigned char BMHD[] = "BMHD";
const unsigned char CMAP[] = "CMAP";
const unsigned char BODY[] = "BODY";
```

If any of these values are identified, then the respective functions are called to retrieve them. The IDs BMHD and CMAP are easily retrieved with fread() calls. However, the image BODY can be compressed using byte-run-length encoding, RLE, or some other mechanism of archiving. This code requires special attention to identify how you are going to retrieve the image bits. The structure declaration for the IFF BitMapHeader is shown in Listing 5-4.

LISTING 5-4 IFF BitMapHeader structure

```
typedef struct
{
    unsigned short  w, h;                   // raster width & height
    unsigned short  x, y;                   // position for this image
    unsigned char   nPlanes;                // # source bitplanes
    unsigned char   masking;                // masking technique
    unsigned char   compression;            // compression algorithm
    unsigned char   pad1;                   // UNUSED. put 0 here.
    unsigned short  transparentColor;       // transparent color number
    unsigned char   xAspect, yAspect;       // aspect ratio
    unsigned short  pageWidth, pageHeight;// size in pixels
} BitMapHeader;

BitMapHeader iffbmHeader;

while(1)
{
  // reading in IFF Chunks
  fread(&chunk,1,sizeof(ChunkHeader),fp);
  chunk.ckSize = intelByteOrderLong( chunk.ckSize );
  if (chunk.ckSize & 1)
    ++chunk.ckSize;

   if(!memcmp(chunk.ckID, BMHD, BYTE_LEN) )
   {
     fread(&iffbmHeader,1,sizeof(BitMapHeader), fp);
   }
   :
}
```

The *BitMapHeader* is very close to the BITMAPINFOHEADER used by the .BMP file format used back in Chapter 2. Figure 5-2 shows a simple IFF file structure and how chunks of data are stored in the *Deluxe Paint II* file.

Now you're getting the idea of what IFF chunks are. The IFF specification states that chunks can be randomly ordered. That is, the BODY chunk may be stored in front of the BMHD chunk. The catch is that the BODY chunk may not

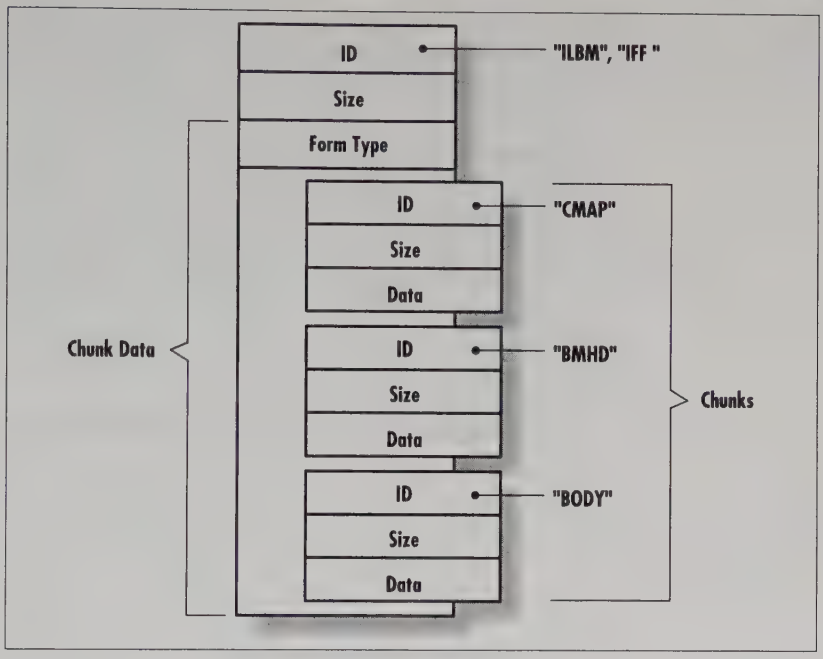


FIGURE 5-2
IFF File structure and data chunks

be processed until the BMHD chunk is read. This explains the usage of the while() loop in the code snippet.

The Microsoft multimedia programming interface uses a form of IFF for waveform storage. This is actually a hybrid flavor IFF, since IFF files were actually designed for a 68000-based processor where the byte ordering is reversed.



RIFF Chunks

A RIFF form chunk is identified by the four-byte identifier "RIFF" or "WAVE". The four-byte identifier "WAVE" represents the Microsoft audio file format for a waveform. Listing 5-5 shows a Visual C++ data structure for a RIFF chunk.

LISTING 5-5 RIFF chunk structure CK

```
typedef  unsigned long DWORD;
typedef unsigned char BYTE;
typedef DWORD FOURCC;

typedef struct {
  FOURCC ckID; // chunk ID
  DWORD ckSize; // chunk size
  BYTE ckData[ckSize]; // chunk data
} CK;
```

This looks exactly like a *ChunkHeader* encountered in the C code presented earlier. The discrepancies between the two are that the Microsoft chunk headers contain buffers for the chunk data. The organization of a RIFF file is shown in Figure 5-3.

Microsoft's *Multimedia Programmer's Reference* states that the IFF four-byte chunk IDs are typedefined as FOURCC (FOUR Character Codes). These are 32-bit (four-byte) identifiers used to tag the data in each specific chunk. The FOURCC definition requires that chunk IDs less than four characters in length be padded with white space characters. An example of padding is the identifier for format, 'fmt', which is three characters long and padded with a trailing *white space*. character. It's useful to define in your code the 32-bit integer equivalences of these four-byte values. This will help speed up comparisons, such as the memcmp() calls.

The FOURCC value is saved in the chunk ID, *ckID* field, of the Microsoft CK data structure. Our code will use this field to see if it understands the FOURCC chunk ID. If it doesn't, it will read the next field in the file, which is the chunk size, *ckSize*, and skip ahead in the file to an offset of *ckSize* bytes.

The *ckSize* field is a 32-bit unsigned integer that states the size of the chunk's data in bytes. The field *ckData* is the physical waveform data. The field *ckData* is WORD-aligned with the start of the file. The original IFF was written for the Motorola 68000 processor which demanded word alignment.



RIFF Forms

A form is an IFF chunk used to encapsulate all the chunks of data in the file. The first four bytes in the file, the form ID, will identify what kind of IFF file

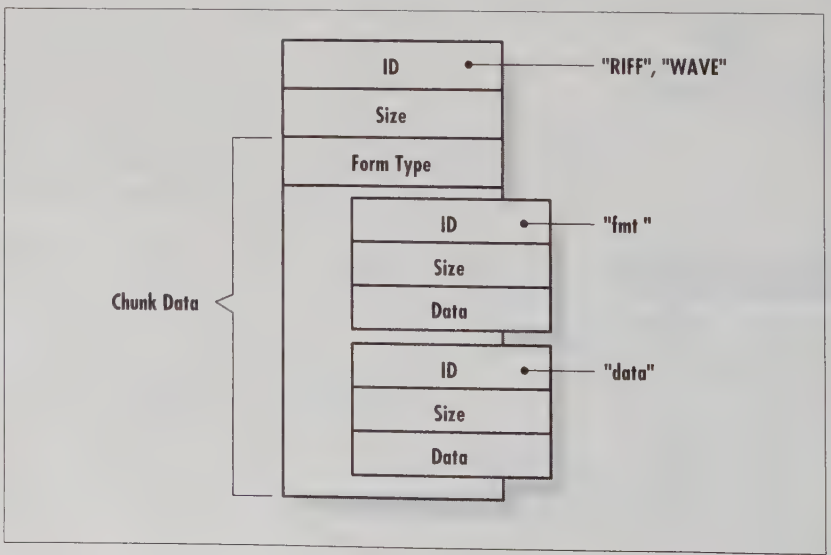


FIGURE 5-3
Organization of a RIFF file



you are working with. Valid form chunk identifiers for Windows 95 sound files are "RIFF" and "WAVE". Located just past the form size (WORD) in the file is a four-character form type tag. Form types for Windows 95 can have the values "RDIB", "RIFF", "RMID", "RMMD", and "WAVE". The form type "RMID" identifies the RIFF MIDI format. These values are located in the first DWORD of form chunk data in the file.

The form type identifies which subchunks are present in the file. Form-type codes in uppercase represent types that have been registered with Electronic Arts and are industry-accepted. Lowercase four-byte codes represent unregistered codes or prototype chunk formats.



The WAVEFORMAT Data Structure

First and foremost, before you can play a WAVE file, you must load it into system memory. We need to specify the physical attributes of the WAVE file, such as the size of the file, the sample rate, and the number of sound channels, to name a few. Sound files can be extremely large files, and any compression schemes used in the storage of the file must also be identified. This information is necessary for determining the required system resources when memory is allocated to load the sound. Information regarding the WAVE format is stored in the RIFF file in specific fields. *The Microsoft Windows SDK Multimedia Reference* uses a notation to identify the RIFF form, the form type, and data chunks. The notation used looks like the following:

```
RIFF ( <formType> <chunk data> ... <chunk data> )
```

The following are some examples of RIFF files:

```
RIFF('WAVE' fmt(1, 1, 11025, 11025, 1, 8)
      data( <wave-data> ) )

RIFF('WAVE' fmt(1, 2, 22050, 44100, 2, 8)
      data( <wave-data> ) )
```

Reading the notation of the RIFF format, you see that both files are RIFF forms containing a WAVE format chunk, 'fmt', and a data chunk, 'data'. The format chunk has its own data structure, WAVEFORMAT, defined in the Windows *Multimedia Programmers SDK*. The WAVEFORMAT structure and its fields are shown in Figure 5-4. The chunk format field contains five fields that are considered to be common among all waveforms that Windows understands.

The fields are stored in successive order in the format chunk. These fields are *wFormatTag*, *nChannels*, *nSamplesPerSec*, *nAvgBytesPerSec*, and *nBlockAlign*. See Figure 5-5 on format chunk organization. Using the second notation example for the RIFF form, the format field looks like this:

```
fmt(1, 2, 22050, 44100, 2, 8)
```

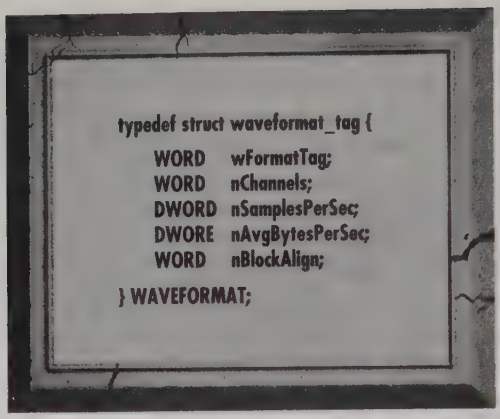


FIGURE 5-4
Format of Waveform data

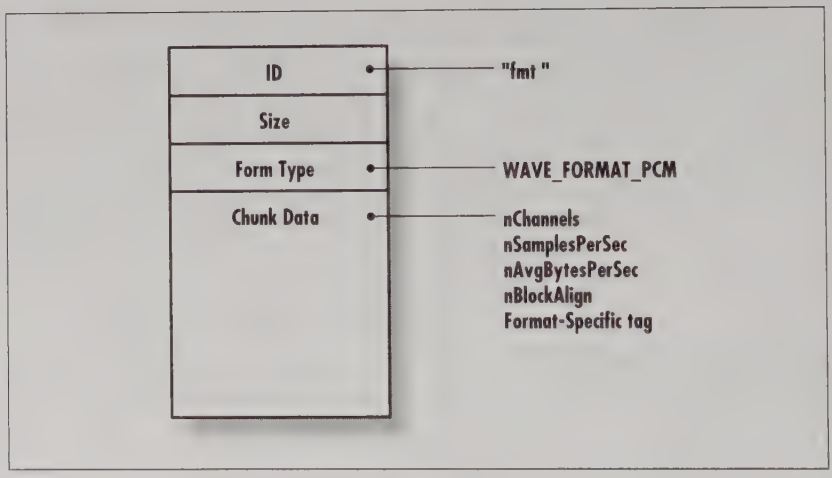


FIGURE 5-5
Data specifiers in the format chunk

Reading these values in from left to right, the WAVEFORMAT field *wFormatTag* has a value of 1. This is defined in a Windows macro as WAVE_FORMAT_PCM. The acronym PCM stands for pulsed code modulation, which I'll discuss in a moment. The second field, *nChannels*, has a value of 2. This indicates the sample is stereo. Had there been a value of 1, as in the first notation example, the sound sample would be monaural (mono). The field *nSamplesPerSec* contains the value 22050. This is read as 22.05 kHZ, the rate at which the sound is sampled. The field following this is *nAvgBytesPerSec*. Its value is 44100. This value is *nSamplesPerSec* times *nChannels*. The second to last field is *nBlockAlign*, for PCM data, which is the number of bytes used by a single sample. The value here is 2 bytes per sample. The last value is called a format specific tag and its value is not written into a field of the WAVEFORMAT structure.

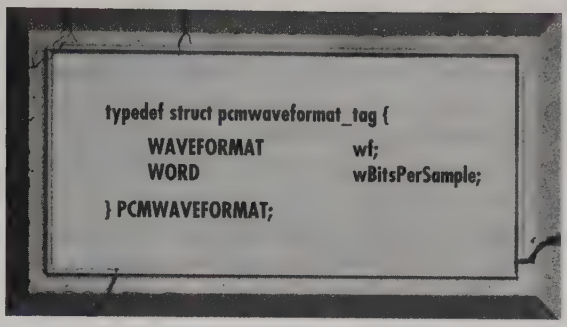


FIGURE 5-6
Data format for PulseCode modulated waveforms



PCMWAVEFORMAT

When the format chunk is parsed, you examine the *wFormatTag* and see that the format type is WAVE_FORMAT_PCM, for PCMFormatSpecific data. The first ten bytes are read into the WAVEFORMAT structure. Using your format chunk notation example, your last value is 8 as you read in the WAVEFORMAT data. The PCM format has an attribute that is not consistent with all Windows waveform data formats. PCM data requires a special tag, *wBitsPerSample* (WORD), to define the number of bits in each sample of each channel. The WAVEFORMAT data and the field *wBitsPerSample* are stored in the PCMWAVEFORMAT structure. Figure 5-6 shows the PCMFormatSpecific waveform data structure.

The PCMWAVEFORMAT structure is used by DirectSound buffer descriptors, DSBUFFERDESC, to create DirectSound buffers. The DirectSound application programmer is responsible for loading the WAVE file and initializing the PCMWAVEFORMAT fields. You have already taken a look at how chunks are stored RIFF files. You understand the format chunk and the data it contains. All you need to do now is put all these concepts together and design some code to unpack RIFF files.



Reading a RIFF File

For your coding example, you will use a simple case where a WAVE file is loaded in as a resource. You'll need two things to start this example, a WAVE file and a resource script file (.RC). The code for this example can be found in the directory for DSTOOLS.



The WAVE Resource

The dstest.rc file requires some specific declarations to identify your WAVE file and its resource ID. The contents of the resource file for DSTOOLS follows.

```
photon          WAV     sftorp.wav
```

The name representing the variable, *lpszSoundName*, "photon," is the string used in the code to refer to the in-memory image of the WAVE file. The next field in the resource file is the string, *"WAV"*. This identifies the type of resource being loaded into memory. The string "sftorp.wav" is the name of the file the resource image will be read in from. For the example, the resource script file is named dstest.rc, but you can name the resource fred.rc, barney.rc, or bambam.rc. The important point to remember is to include this file in your project workspace.

To read the WAVE file image into memory, you have the DSTOOLS function GetWAVRes(). This routine requires a handle to the instance of the executing application that contains the resource. The second parameter is the name of the resource you wish to retrieve. The code for function GetWAVRes() is shown in Listing 5-6.

LISTING 5-6 The GetWAVRes() function

```
void* GetWAVRes(HMODULE hModule, LPCSTR lpszSoundName)
{
  HRSRC   hResInfo;
  HGLOBAL hResData;
  void    *pRIFFBytes;

  if ((hResInfo = FindResource(hModule, lpszSoundName, "WAV"))== NULL)

    return NULL;

  if ((hResData= LoadResource(hModule, hResInfo)) == NULL)
    return NULL;

  if((pRIFFBytes = LockResource(hResData))==NULL)
    return NULL;

  return (void*)pRIFFBytes;
}
```

The first step to retrieving the WAV resource is to assign a handle to reference the resource's info block. The Windows API call, FindResource(), returns a handle to the location of the resource in the specified executing application. The third parameter in FindResource() specifies the resource type. This allows you to locate the WAV resource. Now you need to load the resource into global memory. The Windows API call LoadResource() takes a handle to a system resource and loads it into memory. If the LoadResource() call was successful, it returns a handle from global memory that references the loaded resource. The last step you must make in loading the resource is to look at the WAV resource's global memory. The Win32 call LockResource() accepts the global handle returned from LoadResource() as a calling parameter. If LockResource() is successful, it returns a pointer (void*) to the first byte of the resource in memory.



Win32 calls to LockResource() do not need to Unlock Resources.

An example of using the DSTOOLS routine GetWAVRes() from the generic game application is provided in Listing 5-7.

LISTING 5-7 Calling the GetWAVRes() function from our game application

```
void* pRIFFBytes;

if ((pRIFFBytes = GetWAVRes(NULL,lpResName)) != NULL)
{
  // do something with the WAV resource
}
```

The DSTOOLS function returns a pointer to the first byte in the resource. In terms of RIFF files, this byte is the first byte of the four-byte form chunk ID. Now you are left with the task of unpacking the waveform from the RIFF file.



Unpacking the RIFF Chunks

In the previous section on the RIFF format you examined the composition of form and data chunks. Now you need to pull these concepts together and write a RIFF file unpacker. In this design, the RIFF is in global memory. It was read in as a resource using the DSTOOLS function GetWAVRes(). The function returns a pointer to the first byte in memory of the RIFF file. You are going to pass this pointer to the DSTOOLS function UnpackWAVChunk(). The function UnpackWAVChunk()is shown in Listing 5-8.

LISTING 5-8 The UnpackWAVChunk()function

```
BOOL UnpackWAVChunk( void *pRIFFBytes, LPWAVEFORMATEX *lpwfmx,
                     LPBYTE *lpChunkData, DWORD *lpCkSize )
{
  DWORD *dwChunkBitsPtr;  // current data being referenced
  DWORD *dwChunkTailPtr;  // points to end of chunk
  DWORD dwChunkID;        // four-byte chunk ID
  DWORD dwType;           // form type
  DWORD dwLength;         // size of data in chunk

  // initialize the LPWAVEFORMATEX pointer
  if (lpwfmx)
    *lpwfmx = NULL;

  // initialize the ckData pointer
  if (lpChunkData)
```

continued on next page

continued from previous page

```
    *lpChunkData=NULL;

  // initialize the ckSize pointer
  if (lpCkSize)
    *lpCkSize=0;

  // reference the WAVE resource buffer
  dwChunkBitsPtr = (DWORD*)pRIFFBytes;
  // unpack the chunk ID
  dwChunkID = *dwChunkBitsPtr++;
  // unpack the size field
  dwLength = *dwChunkBitsPtr++;
  // unpack the Form type
  dwType = *dwChunkBitsPtr++;

  // read the 4 byte identifier (FOURCC )
  if (dwChunkID!=mmioFOURCC('R','I','F','F'))
    return FALSE; // not a RIFF

  if (dwType!=mmioFOURCC('W','A','V','E'))
    return FALSE; // not a WAV

  dwChunkTailPtr = (DWORD*)((BYTE*)dwChunkBitsPtr + dwLength-4);

  while(1)
  {
    // unpack the Form Type
    dwType = *dwChunkBitsPtr++;
    // unpack the size
    dwLength = *dwChunkBitsPtr++;

    switch(dwType)
    {
      case mmioFOURCC('f','m','t',' '):
         if (lpwfmx && !*lpwfmx)
         {
           if (dwLength < sizeof(WAVEFORMAT))
             return FALSE; // not WAV

          *lpwfmx = (LPWAVEFORMATEX)dwChunkBitsPtr;

          if ( (!lpChunkData || *lpChunkData)
                    && (!lpCkSize || *lpCkSize))
             return TRUE;

         } // if lpwfmx
      break;  // case 'fmt '

      case mmioFOURCC('d','a','t','a'):
        if ((lpChunkData && !*lpChunkData)
                    || (lpCkSize && !*lpCkSize))
        {
```

```
            if (lpChunkData)
              *lpChunkData = (LPBYTE)dwChunkBitsPtr;

            if (lpCkSize)
              *lpCkSize = dwLength;

            if (!lpwfmx || *lpwfmx)
              return TRUE;
          } // if lpChunkData
        break;
      } // switch dwType
      dwChunkBitsPtr = (DWORD*)((BYTE*)dwChunkBitsPtr
                                      + ((dwLength+1)&~1));
      if (dwChunkBitsPtr >= dwChunkTailPtr)
        break;
    } // while dwChunkBitsPtr
    return FALSE;
}
```

The code is simple if you examine it line by line and read the comments assigned to some of the functions. This is your RIFF file unpacker. The first byte of the pointer to the resource in memory is referencing the form chunk ID. We assign the address of the pointer, *lpRIFFBytes*, to UnpackWAVChunk() pointer *dwChunkBitsPtr*. This is the pointer you will use to walk the bytes of the RIFF stored in global memory. The pointer and chunk declarations are shown in Listing 5-9.

LISTING 5-9 The chunk pointer and data field declarations

```
DWORD *dwChunkBitsPtr;     // current data being referenced
DWORD *dwChunkTailPtr;     // points to end of chunk
DWORD dwChunkID;           // four-byte chunk ID
DWORD dwType;              // form type
DWORD dwLength;            // size of data in chunk

// reference the WAVE resource buffer
dwChunkBitsPtr = (DWORD*)pRIFFBytes;
```

Our first objective is to get the form ID, size, and form type from the RIFF in memory. Figure 5-7 shows the pointer *dwChunkBitsPtr* as it walks the form chunk fields. Each increment of *dwChunkBitsPtr* is an offset of four bytes (DWORD). The code to retrieve the form chunk values is shown in Listing 5-10.

LISTING 5-10 Unpacking the form chunk

```
// unpack the chunk ID
dwChunkID = *dwChunkBitsPtr++;
// unpack the size field
dwLength = *dwChunkBitsPtr++;
// unpack the Form type
dwType = *dwChunkBitsPtr++;
```

The variables *dwChunkID*, *dwLength*, and *dwType* now contain the form chunk data. You need to examine the four bytes of the form ID and form type to make sure you are parsing a RIFF file with a wave format. Listing 5-11 shows the code for reading RIFF form chunk data.

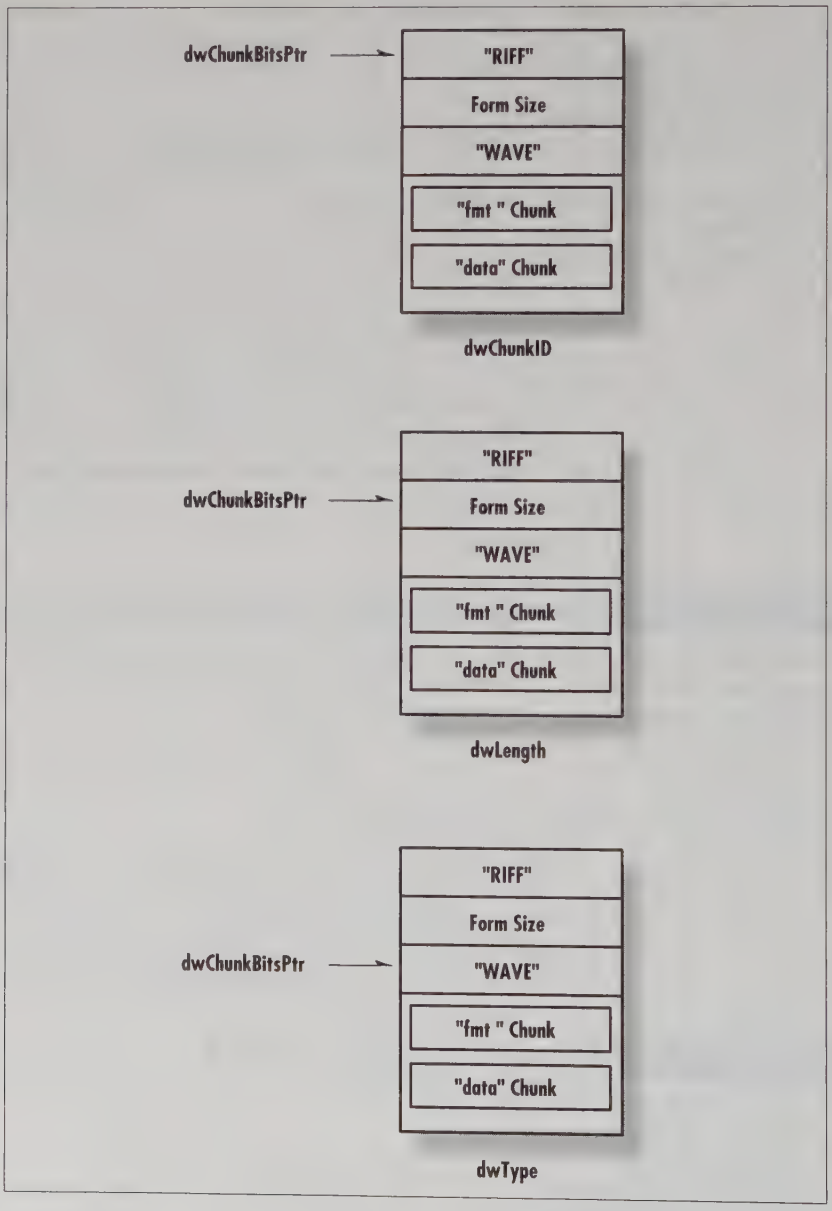


FIGURE 5-7
Reading the values in the form chunk

LISTING 5-11 Unpacking the four-byte 'RIFF' form chunk data

```
// read the 4 byte identifier (FOURCC )
if (dwChunkID!=mmioFOURCC('R','I','F','F'))
        return FALSE; // not a RIFF

if (dwType!=mmioFOURCC('W','A','V','E'))
        return FALSE; // not a WAV

dwChunkTailPtr = (DWORD*)((BYTE*)dwChunkBitsPtr + dwLength-4);
```

You use the Microsoft Windows SDK Multimedia routine FOURCC to compare the values in *dwChunkID* and *dwType* with the desired four-character codes. If, for any reason, these values don't compare, abort the unpacker. The one big drawback to unpackers is that programmers must know exactly what they are looking for. If the RIFF is an unknown form type and the program continues to parse, it will get stuck in an endless loop. This isn't too cool for commercial software. So, if the form type isn't what you're looking for, terminate the unpacking gracefully rather than force the users to reboot their systems or shut down the process.

If you have the correct form type, you need to set the tail pointer to the end of the form. The computation dwLength-4 is used to compute the four-byte offset of the form type which is included in *dwLength* but no longer needed. Our pointer *dwChunkBitsPtr* has already passed the form type field. See Figure 5-7.

With the form ID and form type properly identified you can now begin unpacking the data chunks in the RIFF. The unpacker executes as an endless loop until all chunks have been read. The chunks can be stored in the RIFF in random order.

The premise of unpacking data chunks is that with each iteration of the while loop, your *dwChunkBitsPtr* is referencing a data chunk ID. Figure 5-8 shows the behavior of your pointer as you loop through the data chunks. With each iteration of the loop, you expect to be referencing a new chunk ID or have completely read all the chunks. The pseudo code for reading chunks looks like this:

```
while(1)
{
  // unpack the Form Type
  dwType = *dwChunkBitsPtr++;
  // unpack the size
  dwLength = *dwChunkBitsPtr++;

  switch(dwType)
  {
    case "fmt ":
      // unpack "fmt " chunk
      break;
    case "data":
      //unpack "data" chunk
```

continued on next page

continued from previous page

```
        break;
    } // switch dwType
    dwChunkBitsPtr = (DWORD*)((BYTE*)dwChunkBitsPtr+((dwLength+1)&~1));
    if (dwChunkBitsPtr >= dwChunkTailPtr)
      break;
} // while dwChunkBitsPtr
```

The fields *dwType* and *dwLength* contain the chunk four-byte ID and chunk size, respectively. A case statement is used to identify what chunk you are about to unpack. The chunk ID will either be "fmt" or "data".



Unpacking Format Chunks

The format chunk contains waveform data on how many channels the sample is and the sampling rate. This information is stored in the Windows structure WAVEFORMATEX. A check is made to ensure that chunk size is not smaller than the size of a WAVEFORMAT structure. If it is, the data contained in it is insufficient to define a waveform. The code for unpacking the format chunk is shown in Listing 5-12.

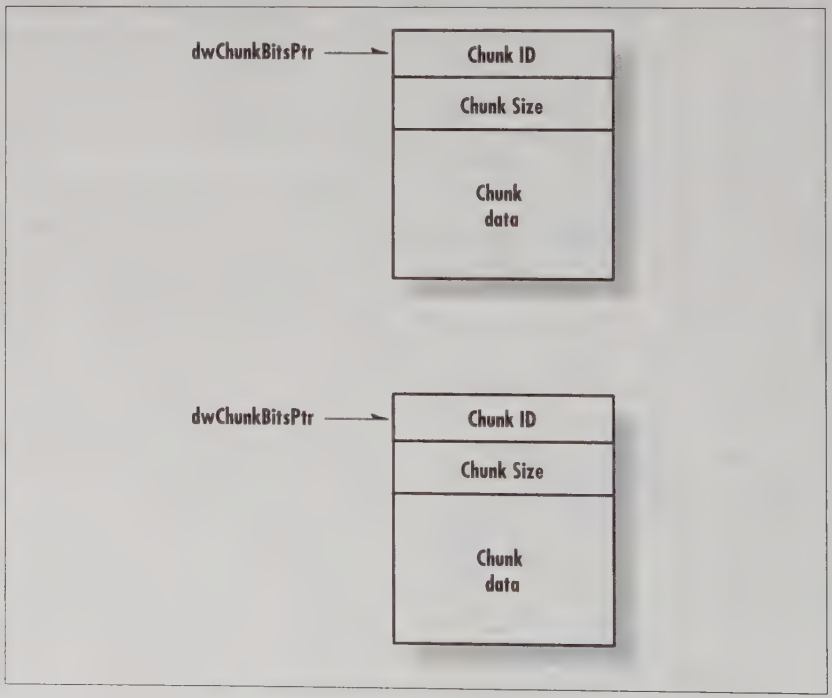


FIGURE 5-8
Unpacking the chunk ID and size

LISTING 5-12 Unpacking the four-byte 'fmt ' chunk

```
case mmioFOURCC('f','m','t',' '):
  if (lpwfmx && !*lpwfmx)
  {
    if (dwLength < sizeof(WAVEFORMAT))
      return FALSE; // not WAV

    *lpwfmx = (LPWAVEFORMATEX)dwChunkBitsPtr;

    if ( (!lpChunkData || *lpChunkData) && (!lpCkSize || *lpCkSize))
      return TRUE;
  } // if lpwfmx
  break;  // case 'fmt '
```

This is the chunk you need to inform DirectSound what kind of sound surface you want to create. Without this information the samples in the chunk data are useless. With the number of channels, sampling rate, and format type identified, you can look for the data chunk and read it in.



Unpacking the Waveform Data Chunk

When the four-byte ID, "data", of the data chunk has been located in the form, you can start unpacking the digitized sample. In order to play back the sample in DirectSound, you need to provide two external buffer pointers. One pointer, *lpChunkData*, references the first byte of sample data. The other pointer, *lpChunkSize*, references the value for the length of the sample bytes. The snippet of code to unpack the data chunk looks like Listing 5-13 .

LISTING 5-13 Unpacking the four-byte 'data' chunk

```
case mmioFOURCC('d','a','t','a'):
  if ((lpChunkData && !*lpChunkData) || (lpCkSize && !*lpCkSize))
  {
    if (lpChunkData)
      *lpChunkData = (LPBYTE)dwChunkBitsPtr;

    if (lpCkSize)
      *lpCkSize = dwLength;

    if (!lpwfmx || *lpwfmx)
      return TRUE;
  } // if lpChunkData
  break;
```

In this code you assign the pointer *lpChunkdata* to reference the current location in the data chunk, which is the current address of *dwChunkBitsPtr*. The length of the chunk, *dwLength*, is stored in a variable external to this function. Notice that one of the conditions for breaking your while() loop is to test to see if you already unpacked the format chunk. You performed a similar check when you unpacked the format data to see if you already unpacked the sample data. You do this because the chunks may not be stored in order in the form chunk.

If the unpacking of the format and data chunks are successful, the routine UnpackWAVChunk() returns a Boolean value of TRUE. This indicates that you have successfully retrieved the sample data, the size of the sample data, and the fields of the WAVEFORMAT structure. All you have left to do now is to set up the DirectSound routines to play the waveforms.



The DirectSound Interface

To play your waveform you need to instantiate a DirectSound object. DirectSound architecture uses the COM and HAL interfaces explained in Chapter 3. Like DirectDraw objects, only one DirectSound object may exist per sound device available on the calling system. However, multiple DirectSound instances may exist per DirectSound object. Once a DirectSound object is created it may be accessed through the IDirectSound interfaces.



Creating a DirectSound Object

A routine is provided in DSTOOLS to create the DirectSound object for us. This function is called DSInit(), for DirectSound initialize. The routine calls the DirectSound function DirectSoundCreate(). For this call, pass a NULL value for the GUID, an uninitialized pointer to the DirectSound object, and another NULL value. The first NULL value is the GUID to the DirectSound device driver. A value of NULL represents the default device driver present in the system. Figure 5-9 shows the DirectSound API call for DirectSoundCreate().

Just as you did for creating the DirectDraw object, you must set the cooperative level for this instance of DirectSound. When the application focus is set, the Windows audio resources can be assigned exclusively to this instance. If the cooperative level is not set, the sound buffers can't be played.

That's pretty much all you have to do to create a DirectSound object. One word of warning from the DirectSound documentation is that a WAVE API instance and DirectSound can't share the same audio device. If a WAVE API

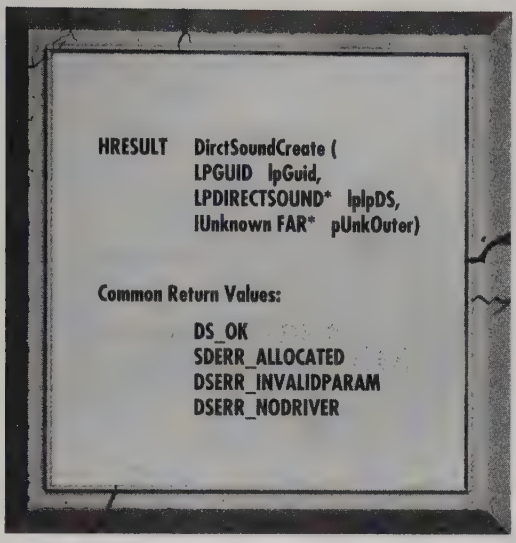


FIGURE 5-9

Creating a DirectSound object

routine is using the sound device when the DirectSoundCreate() function is called, the DirectSound routine will fail. The code used by DSTOOLS.CPP to create a DirectSound object is shown in Listing 5-14.

LISTING 5-14 Creating the DirectSound object

```
LPDIRECTSOUND lpDS;

BOOL DSInit( HWND hwnd )
{
  HRESULT dsval;

  dsval = DirectSoundCreate( NULL, &lpDS, NULL );
  if (dsval != DS_OK)
    return DSError("DirectSoundCreate FAILED");

  dsval = lpDS->SetCooperativeLevel( hwnd, DSSCL_NORMAL );
  if (dsval != DS_OK)
    return DSError("SetCooperativeLevel FAILED");
  return TRUE;
}
```

The function DSError(), used by DSInit(), is a simple error handler to alert the user when any of the DirectSound API calls should fail. The code for DSError() is shown in Listing 5-15.

LISTING 5-15 *Spells of Fury* DSError() utility

```
BOOL DSError( char *errMsg )
{
  char errMsgBuf[80];

  wsprintf(errMsgBuf, errMsg);
  MessageBox(GetActiveWindow(), errMsgBuf,
                  "DirectSound Error", MB_OK);
  return FALSE;
}
```

These routines provide a DirectSound object you can use to play your waveforms on. You now need functions to create DirectSound buffers, write waveforms to them, and then play them. The next section looks at how to create and write DirectSound buffers.



Creating DirectSound Buffers

To create a DirectSound buffer, you need to initialize the fields of the DirectSound DSBUFFERDESC structure. Figure 5-10 shows the data structure to define a DirectSound buffer. Once these fields have been filled in correctly, the application can call the DirectSound member CreateSoundBuffer(). Your RIFF unpacker already returns the values you need to create the DirectSound buffer. You'll notice from Figure 5-10 that the DSBUFFERDESC structure has a LPWAVEFORMATEX field and a *dwBufferBytes* field for the size of the wave sample. Listing 5-16 shows how to initialize the DSBUFFERDESC fields for the DirectSound buffer.

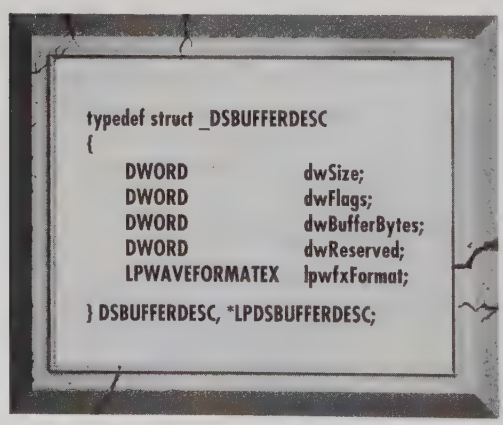


FIGURE 5-10
Describing the DirectSound buffer

LISTING 5-16 Creating a DirectSound buffer

```
LPCSTR              lpResName;
LPDIRECTSOUNDBUFFER lpDSB = NULL;
DSBUFFERDESC        dsbd;
LPBYTE              lpWaveData;
HRESULT             dsval;
void                *pRIFFBytes;

memset(&dsbd, 0x00, sizeof(DSBUFFERDESC));
if ((pRIFFBytes = GetWAVRes(NULL,lpResName)) != NULL)
{
  UnpackWAVChunk(pRIFFBytes,
            &dsbd.lpwfxFormat, lpWaveData, &dsbd.dwBufferBytes);

  dsbd.dwSize = sizeof(dsbd);
  dsbd.dwFlags = DSBCAPS_STATIC | DSBCAPS_CTRLDEFAULT;
  dsval = lpDS->CreateSoundBuffer( &dsbd, &lpDSB, NULL );
}
```

This code demonstrates how the DSTOOLS routines can be used to create a DirectSound buffer. The DSTOOLS call, GetWAVRes(), loads the resource into global memory and assigns the pointer *lpRIFFBytes* to the first byte of the RIFF form. The pointer, *lpRIFFBytes*, is handed off to UnpackWAVChunk(). For calling arguments you pass the LPWAVEFORMATEX field of the DSBUFFERDESC, a pointer, *lpWaveData*, to reference the bytes when they get unpacked, and another field from DSBUFFERDESC, *dwBufferBytes*, that tells DirectSound how large the waveform in buffer *lpWaveData* is. When the DSTOOLS function UnpackWAVChunk() has completed, the remaining fields of the DSBUFFERDESC structure are initialized. When all the necessary values of the DSBUFFERDESC structure have been filled in, the DirectSound member CreateSoundBuffer() is called. A description of CreateSoundBuffer() is in Figure 5-11.

The DirectSound buffer you have just created doesn't contain any data. It's an empty buffer allocated by DirectSound to meet the needs of the waveform you described with the DSBUFFERDESC structure. Now you need to write the waveform data into the buffer.



Writing Waveforms to a DirectSound Buffer

The DirectSound API doesn't provide any tools to write raw data into a DirectSound buffer so you're going to need to add one for your own use. The DSTOOLS routine contains a function, WriteWAVData(), to write the RIFF bytes into a DirectSound buffer. The DSTOOLS function WriteWAVData() is shown in Listing 5-17.

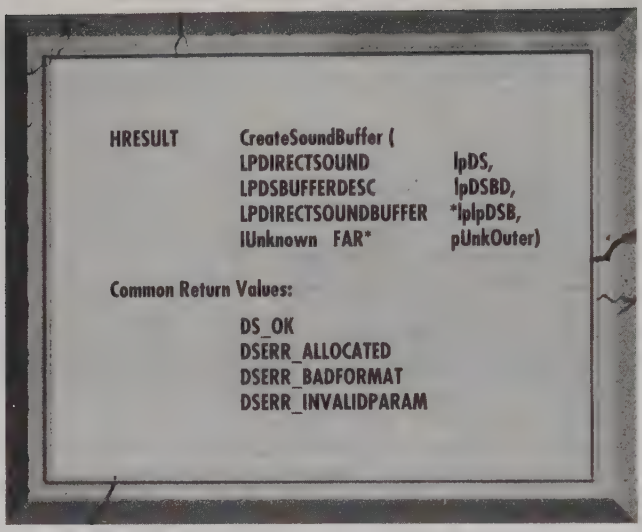


FIGURE 5-11
The DirectSound member to create sound buffers

LISTING 5-17 Writing to a DirectSound buffer

```
BOOL WriteWAVData( LPDIRECTSOUNDBUFFER lpDSB, LPBYTE lpWaveData,
                                              DWORD dwWriteBytes )
{
  HRESULT dsval;

  if (lpDSB && lpWaveData && dwWriteBytes)
  {
    LPVOID lpAudioPtr1, lpAudioPtr2;
    DWORD dwAudioBytes1, dwAudioBytes2;

    dsval = lpDSB->Lock(0, dwWriteBytes,
                    &lpAudioPtr1, &dwAudioBytes1,
                    &lpAudioPtr2, &dwAudioBytes2, 0);
    if (dsval == DS_OK)
    {
      CopyMemory(lpAudioPtr1, lpWaveData, dwAudioBytes1);

      if( dwAudioBytes2 != 0)
        CopyMemory(lpAudioPtr2, lpWaveData + dwAudioBytes1,
                                                dwAudioBytes2);

      lpDSB->Unlock(lpAudioPtr1, dwAudioBytes1,
                          lpAudioPtr2, dwAudioBytes2);
      return TRUE;
    }
  }
  return FALSE;
}
```

For a wave byte memory to DirectSound buffer copy routine, this seems like a lot of code. The reason is that your wave bytes are stored in linear system memory and the DirectSound memory is circular. To write your waveform data to a DirectSound buffer, you must use the DirectSound member Lock(), which will return a pointer to allow you to write to this memory. Figure 5-12 shows the DirectSound interface for locking a sound buffer.

When DirectSound locks a DirectSound buffer, the member Lock() actually returns two write pointers. A sound buffer can be static (where it contains the entire sound), or it can be streaming (where it contains only part of the sound). In a streaming buffer, you are faced with the situation of writing to the buffer while it is playing. With streaming buffers, the situation could arise where an application locks the buffer to write 600 bytes, starting 300 bytes into the buffer. Listing 5-18 shows an example of using the DirectSoundBuffer member Lock() for writing to a streaming buffer.

LISTING 5-18 Writing to a streaming buffer using DirectSound

```
LPVOID lpAudioPtr1, lpAudioPtr2;
DWORD dwAudioBytes1, dwAudioBytes2;

dsval = lpDSB->Lock(0, dwWriteBytes, &lpAudioPtr1, &dwAudioBytes1,
                                     &lpAudioPtr2, &dwAudioBytes2, 0);
```

In the example, the variable *dwWriteBytes* contains the value 600, which is the number of bytes that you wish to write to the buffer. The lock is attempted on the buffer 300 bytes into it. The DirectSound member, Lock(), will return two pointers, *lpAudioPtr1* and *lpAudiPtr2*. The pointer *lpAudioPtr1* references the bytes 300 through 799 in the stream buffer. This is only 500 bytes (*dwAudioBytes1*) and you need 600. The second pointer, *lpAudioPtr2*, references the first 100 bytes (*dwAudioBytes2*), as shown in Figure 5-13.

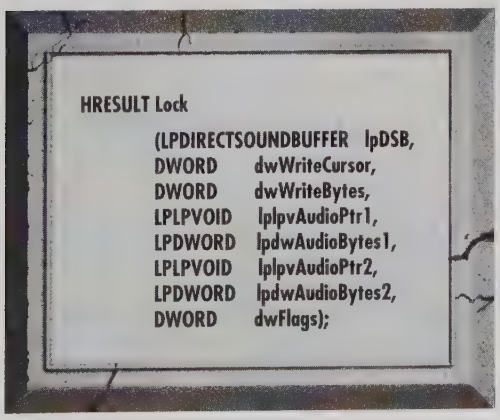


FIGURE 5-12
Interface to lock DirectSound buffers

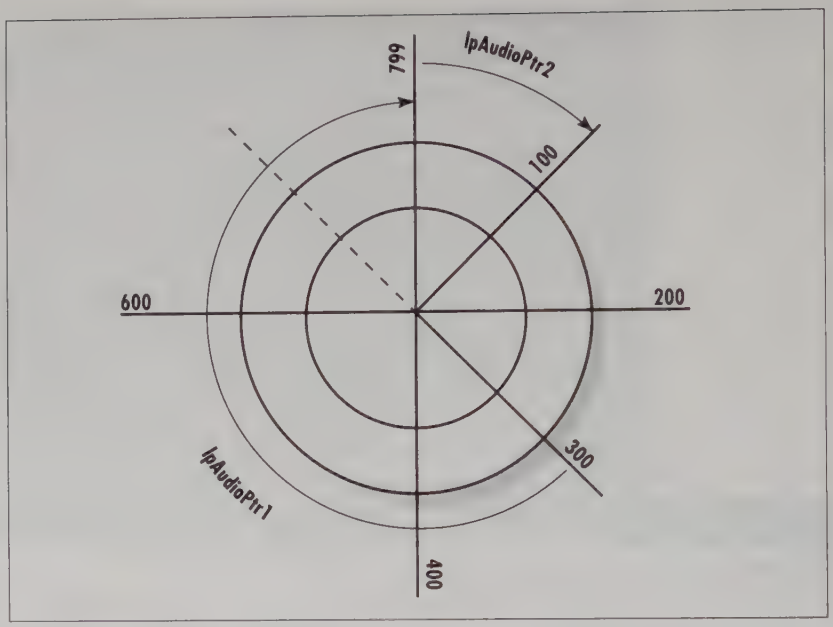


FIGURE 5-13
Locking streaming DirectSound buffers

If the DirectSound Lock() occurs 100 bytes into the buffer, the pointer *lpAudioPtr1*, has a sufficient amount of bytes to completely write the sound data. In this case, the second pointer, *lpAudioPtr2*, will be NULL and the value at *&dwAudioBytes2* will be 0.

If the DirectSound member Lock() is successful, you may proceed with writing your data into the buffer. Using the Win32 block copy function, CopyMemory(), you can write the waveform into the DirectSound buffer. The code to write the data into the buffer is provided in Listing 5-19.

LISTING 5-19 Writing WAV data to a DirectSound buffer

```
if (dsval == DS_OK)
{
  CopyMemory(lpAudioPtr1, lpWaveData, dwAudioBytes1);

  if( dwAudioBytes2 != 0)
    CopyMemory(lpAudioPtr2,
       lpWaveData + dwAudioBytes1, dwAudioBytes2);

  lpDSB->Unlock(lpAudioPtr1,
       dwAudioBytes1, lpAudioPtr2, dwAudioBytes2);
  return TRUE;
}
```

At the most, you are required to perform two memory writes. If the value of *dwAudioBytes2* is 0, you have a case where *lpAudioPtr1* referenced a sufficient

amount of memory to write the data into the sound buffer. If the value of *dwAudioBytes2* is non-zero, then you must add the fragment to your original data, *lpWaveData* + *dwAudioBytes1*, with a byte length of *dwAudioBytes2*, to the memory referenced by *lpAudioPtr2*. Upon completion, release the locked DirectSound buffer using the member Unlock(). A feature of Unlock() that requires some attention is the fact that there are two pointers, one to the buffer written, the second to the address of the number of bytes written. The DirectSound member Unlock() is shown in Figure 5-14.

In most applications you will want to load the WAVE resource file, unpack the RIFF in global memory, create a DirectSound buffer for the data, and then write the data to the DirectSound buffer. The DSTOOLS routine MakeSoundBuffer() provides this functionality. It makes and loads a DirectSound buffer based on the WAVE resource name passed to it. (See Listing 5-20.)

LISTING 5-20 Creating a DirectSound buffer from a WAV resource

```
IDirectSoundBuffer* MakeSoundBuffer( LPDIRECTSOUND lpDS, LPCSTR lpResName)
{
  LPDIRECTSOUNDBUFFER lpDSB;
  DSBUFFERDESC        dsbd;
  LPBYTE              lpWaveData;
  HRESULT             dsval;
  void                *pRIFFBytes;

  lpDSB = NULL;
  memset(&dsbd, 0x00, sizeof(DSBUFFERDESC));
  if ((pRIFFBytes = GetWAVRes(NULL,lpResName)) != NULL)
  {
    UnpackWAVChunk(pRIFFBytes,
       &dsbd.lpwfxFormat, &lpWaveData, &dsbd.dwBufferBytes);

    dsbd.dwSize = sizeof(dsbd);
    dsbd.dwFlags = DSBCAPS_STATIC | DSBCAPS_CTRLDEFAULT;
    dsval = lpDS->CreateSoundBuffer( &dsbd, &lpDSB, NULL );
    if (dsval == DS_OK)
    {
      if (WriteWAVData(lpDSB,
              lpWaveData, dsbd.dwBufferBytes) == FALSE)
      {
        lpDSB->Release;
        lpDSB = NULL;
      }
    } // if DS_OK
    else
    {
      lpDSB = NULL;
    } // else
  } // if GetWAVRes
  return (IDirectSoundBuffer*)lpDSB;
}
```

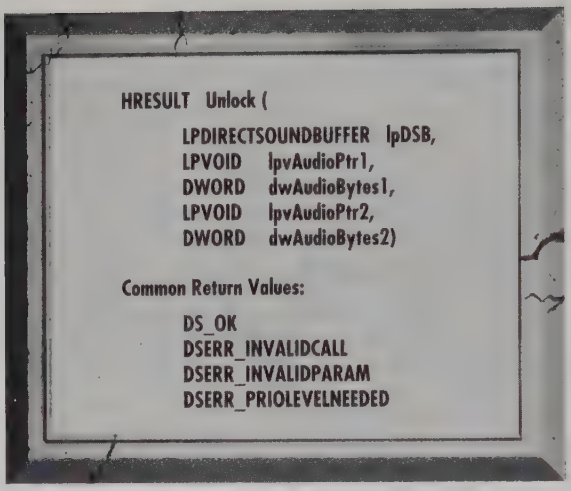


FIGURE 5-14
DirectSound member to unlock a buffer after a write

The DSTOOLS function creates the DirectSound buffer and writes the data from the RIFF into it. You're getting closer to the point where you can start playing some DirectSound buffers, but you still have some work ahead of you. In the next section you'll look at designing a data structure for DSTOOLS to maintain information about various sounds you want to use in your games.



Game Sound Data Structures and DirectSound

The numerous types of sounds you can add to your games is only limited by your imagination. Even though the possibilities are vast, the amount of information you need to maintain about your sounds isn't. The WAVE sounds you have looked at so far have attributes that are present in both the RIFF form and the DirectSound DSBUFFERDESC. These include the Windows LPWAVEFORMATEX structure, a buffer for the raw data, a buffer size, and a DirectSound buffer associated to the waveform. Listing 5-21 shows the declaration of a structure, WAVINFO, that you'll use to define your game application's sound data .

LISTING 5-21 *Spells of Fury* Sound WAVINFO structure

```
typedef struct {
  LPBYTE              lpWAVData;
  DWORD               dwWAVSize;
  int                 bufCount;
  int                 bufferIndex;
  LPWAVEFORMATEX      lpwfmx;
  LPDIRECTSOUNDBUFFER lpDSBuf[1];
} WAVINFO, *LPWAVINFO;
```



Using WAVEINFO Structures

You're going to build up your DirectSound tools around the WAVEINFO data structure you have created. When you are finished, you will be able to play a sound just by passing a LPWAVINFO pointer to a DSTOOLS function, PlayWAVInfo(). Building on the tools you already have, such as MakeSoundBuffer() and GetWAVRes(), you can initialize your WAVEINFO data structure using the function CreateWAVInfo(), shown in Listing 5-22.

LISTING 5-22 *Spells of Fury* **CreateWAVInfo() function**

```
LPWAVINFO CreateWAVInfo(LPDIRECTSOUND lpDS, LPSTR lpResName, int bufCount )
{
  LPWAVINFO lpwi;
  void*     pRIFFBytes;
  int       wiSize;

  if (bufCount < 1)
    bufCount = 1;

  wiSize = sizeof(WAVINFO)
       + (bufCount - 1)*sizeof(LPDIRECTSOUNDBUFFER);
  lpwi = (LPWAVINFO)LocalAlloc(LPTR, wiSize);
  if (lpwi == NULL)
    return NULL;

  lpwi->bufCount = bufCount;

  if((pRIFFBytes = GetWAVRes(NULL,lpResName)) != NULL)
  {
    UnpackWAVChunk(pRIFFBytes,
              &lpwi->lpwfmx,&lpwi->lpWAVData,&lpwi->dwWAVSize);

    int i;
    for (i=0; i<lpwi->bufCount; i++)
    {
      HRESULT dsval;
      dsval = lpDS->DuplicateSoundBuffer(lpwi->lpDSBuf[0],
                                         &lpwi->lpDSBuf[i]);
      if (dsval != DS_OK)
      {
        lpwi->lpDSBuf[i] = MakeSoundBuffer(lpDS, lpResName);
        if (!lpwi->lpDSBuf[i])
        {
          FreeWAVInfo(lpwi);
          lpwi=NULL;
          break;
        }
      } // if dsval != DS_OK
```

continued on next page

continued from previous page

```
    } // for i
  } // if GetWAVRes
  return lpwi;
}
```

In your code, you will reference WAVEINFO structures through pointers, using LPWAVINFO for referencing. You'll need to allocate system memory so that you can hold all of your waveform data, waveform size, LPWAVEFORMATEX structure, and an array of pointers to DirectSound buffers. The array of pointers will allow you to play sounds simultaneously. An example would be a rapid-fire laser cannon on a *Defender* game. You could have as many as 30 shots, each with a phaser sound in some mode of play. So the array of pointers for DirectSound buffers will reference 1 to *n* buffers. Initialization of LPWAVINFO memory is shown in Listing 5-23.

LISTING 5-23 Allocating LPWAVINFO memory

```
LPWAVINFO lpwi;
void*     pRIFFBytes;
int       wiSize;

if (bufCount < 1)
  bufCount = 1;
wiSize = sizeof(WAVINFO) + (bufCount -1)*sizeof(LPDIRECTSOUNDBUFFER);
lpwi = (LPWAVINFO)LocalAlloc(LPTR, wiSize);
if (lpwi == NULL)
  return NULL;
lpwi->bufCount = bufCount;
```

Determine the size of the LPWAVINFO buffer you need to allocate by adding the size of the WAVEINFO structure plus the size of the LPDIRECTSOUNDBUFFER times the number of buffers you want. The number of buffers you create you will store in the WAVEINFO structure field, *bufCount*.

If your LPWAVINFO buffer allocation is successful, you can proceed with the familiar call to GetWAVRes() to load the RIFF form bytes into global memory. If the RIFF form makes it into memory with no problems, you commence the RIFF chunk unpacking. (See Listing 5-24.)

LISTING 5-24 Loading the RIFF data into the LPWAVINFO buffer

```
if((pRIFFBytes = GetWAVRes(NULL,lpResName)) != NULL)
{
  UnpackWAVChunk(pRIFFBytes,&lpwi->lpwfmx,
            &lpwi->lpWAVData,&lpwi->dwWAVSize);
  int i;
  for (i=0; i<lpwi->bufCount; i++)
  {
```

```
    // copy waveform bytes to all DirectSound buffers
  } // for i

} // if GetWAVRes
```

Concluding the RIFF unpacking exercise, you will do something a little bit different than you did in the previous example. The WAVEINFO field *bufCount* contains a value of how many buffers you want to be able to play simultaneously. The *for* loop in the code above has a comment where something miraculous occurs. There is nothing close to voodoo programming going on here. What you are doing is copying the waveform data into every instance of DirectSound buffers associated to this LPWAVINFO buffer, shown in Listing 5-25.

LISTING 5-25 Creating simultaneous sound buffers

```
for (i=0; i<lpwi->bufCount; i++)
{
  HRESULT dsval;
  dsval = lpDS->DuplicateSoundBuffer(lpwi->lpDSBuf[0],
                                     &lpwi->lpDSBuf[i]);
  if (dsval != DS_OK)
  {
    lpwi->lpDSBuf[i] = MakeSoundBuffer(lpDS, lpResName);
    if (!lpwi->lpDSBuf[i])
    {
      FreeWAVInfo(lpwi);
      lpwi=NULL;
      break;
    }
  } // if dsval != DS_OK
} // for i
```

What you are doing is looping through the LPDIRECTSOUNDBUFFER array that you have indexed up to a maximum of *bufCount* entries. Doing this takes a DirectSound buffer and creates new DirectSound buffer objects which reference the data in the original buffer. You are assigning multiple buffers to reference the same DirectSound buffer memory as the original. In this case, *lpwi->lpDSBuf[0]* is your master buffer memory and all other buffers use this buffer memory when they play their sound. The DirectSound member DuplicateSoundBuffer() is shown Figure 5-15.

After you have completed your DirectSound buffer cloning process, pass the buffer on to one of your DSTOOLS routines, MakeSoundBuffer(). This is the routine you have been using to read the WAVE resource into global memory. Do your magic hand waving, and out comes a DirectSound buffer loaded with the wave data from the resource.

In the event your DSTOOLS routine, MakeSoundBuffer(), fails, you need to free the resources used to describe the WAVEINFO structure. The DSTOOLS routine for this is FreeWAVInfo() and its code is shown in Listing 5-26.

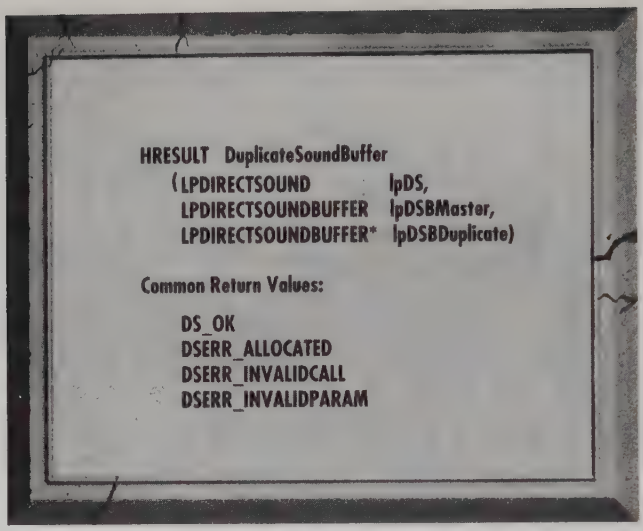


FIGURE 5-15
Creating DirectSound buffer objects that use the same memory

LISTING 5-26 Freeing the *Spells of Fury* LPWAVINFO buffer

```
void FreeWAVInfo(LPWAVINFO lpwi)
{
  if(lpwi)
  {
    int i;
    for(i=0; i<lpwi->bufCount; i++)
    {
      if (lpwi->lpDSBuf[i])
      {
        lpwi->lpDSBuf[i]->Release();
        lpwi->lpDSBuf[i] = NULL;
      }  // if lpwi->lpDSBuf[i]
    } // for i
    LocalFree(lpwi);
  } // if lpwi
}
```

The function basically frees all the DirectSound buffers you allocated previously and then deallocates the LPWAVINFO buffer. This is your mechanism for shutting down your WAVEINFO data structure when you run into an unrecoverable problem with memory allocation or DirectSound resources.



Playing WAVEINFO Data with DirectSound

You're at the brink of being able to play your waveforms with DirectSound. In this section, you'll look at how to play a sound stored in the WAVEINFO

structures. The routine in DSTOOLS, PlayWAVInfo(), is what you use to play the WAVEINFO sounds. The code for PlayWAVInfo() is shown in Listing 5-27.

LISTING 5-27 Playing DirectSound using *Spells of Fury* PlayWAVInfo()

```
BOOL PlayWAVInfo(LPWAVINFO lpwi, DWORD dwPlayFlags)
{
  HRESULT dsval;

  if (lpwi == NULL)
    return FALSE;

  if ((!(dwPlayFlags & DSBPLAY_LOOPING) || (lpwi->bufCount == 1)))
  {
    LPDIRECTSOUNDBUFFER lpDSB = GetAvailableBuf(lpwi);
    if (lpDSB != NULL)
    {
      dsval = lpDSB->Play(0,0,dwPlayFlags);
    } // if lpwi != NULL
  } // if dwPlayFlags

  if (dsval == TRUE)
    return TRUE;
  else
    return FALSE;
}
```

The function PlayWAVInfo() plays a waveform looping (DSBPLAY_LOOPING) or for a single instance. The routine calls the DSTOOLS function GetAvailableBuf(). Remember, you can have more than one buffer playing at a time. The trick is to determine which buffer is available to play a sound. The code for GetAvailableBuf() is shown in Listing 5-28.

LISTING 5-28 Locating an available DirectSound buffer

```
IDirectSoundBuffer *GetAvailableBuf(LPWAVINFO lpwi)
{
  LPDIRECTSOUNDBUFFER lpDSB;
  HRESULT             hres;
  DWORD               dwStatus;

  if (lpwi ==NULL)
    return NULL;

  if (lpDSB = lpwi->lpDSBuf[lpwi->bufferIndex])
  {
    hres = lpDSB->GetStatus(&dwStatus);
    if (hres != DS_OK)
      dwStatus = 0;
```

continued on next page

continued from previous page

```
    if ((dwStatus & DSBSTATUS_PLAYING) == DSBSTATUS_PLAYING)
    {
      if (lpwi->bufCount > 1)
      {
        if (++lpwi->bufferIndex >= lpwi->bufCount)
          lpwi->bufferIndex = 0;
        lpDSB = lpwi->lpDSBuf[lpwi->bufferIndex];
        hres = lpDSB->GetStatus(&dwStatus);

        if ((hres == DS_OK)&&(dwStatus & DSBSTATUS_PLAYING)
                                          == DSBSTATUS_PLAYING)
        {
          lpDSB->Stop();
          lpDSB->SetCurrentPosition(0);
        } // if hres && dwStatus
      } // if lpwi->bufCount
      else
        lpDSB = NULL;
    } // if dwStatus == DSBSTATUS_PLAYING

    if (lpDSB && (dwStatus & DSBSTATUS_BUFFERLOST))
    {
      hres = lpDSB->Restore();
      if ((hres != DS_OK)
         || !WriteWAVData(lpDSB, lpwi->lpWAVData, lpwi->dwWAVSize))
      {
        lpDSB == NULL;
      }
    } // if lpDSB && DSBSTATUS_BUFFERLOST
  } // if lpDSB->Buffers[lpwi->bufferIndex]
  return lpDSB;
}
```

A WAVEINFO sound can have 1 to *n* buffers. The function GetAvailableBuf() cycles through the buffers, incrementing its index, *bufferIndex*, by one each time. If the current DirectSound buffer (*lpwi->lpDSBuf[bufferIndex]*) is not NULL, check to see if the buffer is currently playing a sound. The DirectSound member GetStatus(), shown in Figure 5-16, is used to determine if a buffer is currently being played. If the buffer is currently playing, you now descend into this portion of the code shown in Listing 5-29.

LISTING 5-29 Determining if a buffer is in use

```
if ((dwStatus & DSBSTATUS_PLAYING) == DSBSTATUS_PLAYING)
{
  if (lpwi->bufCount > 1)
  {
```

```
    if (++lpwi->bufferIndex >= lpwi->bufCount)
      lpwi->bufferIndex = 0;
    lpDSB = lpwi->lpDSBuf[lpwi->bufferIndex];
    hres = lpDSB->GetStatus(&dwStatus);
    if ((hres == DS_OK)&&(dwStatus & DSBSTATUS_PLAYING)
                                    == DSBSTATUS_PLAYING)
    {
      lpDSB->Stop();
      lpDSB->SetCurrentPosition(0);
    } // if hres && dwStatus
  } // if lpwi->bufCount
  else
    lpDSB = NULL;
} // if dwStatus == DSBSTATUS_PLAYING
```

If a DirectSound buffer is currently playing, and the LPWAVINFO header has more than one buffer available, *bufferIndex* will be incremented to the next buffer. The DirectSound member GetStatus() is called again to determine if the current buffer is playing a sound. If it is, the current sound playing in this buffer is stopped. If you are stopping a buffer from playing, you are assuming that you have cycled through in the number of available buffers. The DirectSound member SetCurrentPosition() is called to start the buffer playing from the beginning. The DirectSound member to set the buffer cursor is shown in Figure 5-17.

The other condition that the DirectSound documentation recommends you check for is if your DirectSound buffer is lost. The DSTOOLS routine GetAvailableBuf() uses this to determine if a buffer is available or has been lost. To check to see if a DirectSound buffer needs to be restored, see Listing 5-30.

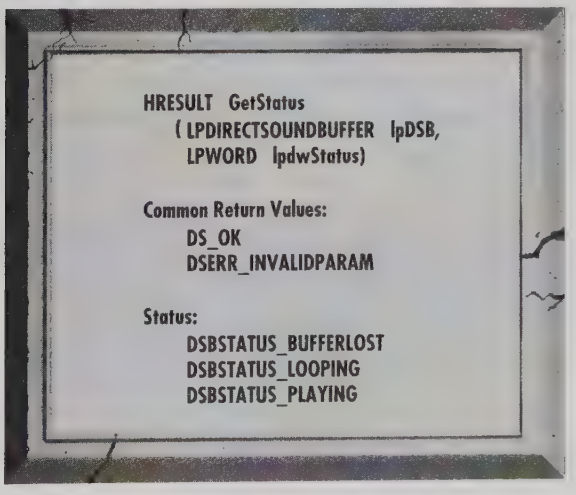


FIGURE 5-16

DirectSound member to get the buffer status

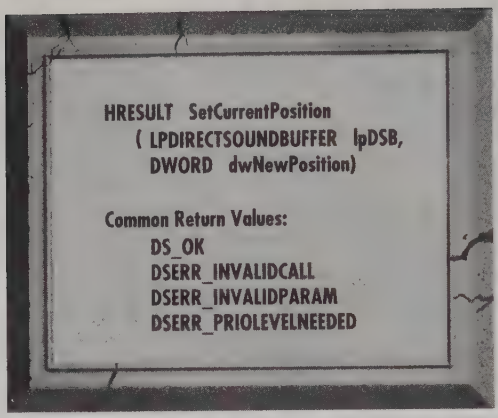


FIGURE 5-17
DirectSound member to set the buffer cursor position

LISTING 5-30 Restoring lost DirectSound buffers

```
if (lpDSB && (dwStatus & DSBSTATUS_BUFFERLOST))
{
  hres = lpDSB->Restore();
  if ((hres != DS_OK)
       ||!WriteWAVData(lpDSB, lpwi->lpWAVData, lpwi->dwWAVSize))
  {
  lpDSB == NULL;
  }
} // if lpDSB && DSBSTATUS_BUFFERLOST
```

If the DirectSound buffer is lost, based on the results of GetStatus(), reload the DirectSound buffer with the waveform data stored in the WAVEINFO structure. The DSTOOLS WriteWAVData() can be called to write the wave data into the DirectSound buffer.

Once GetAvailableBuf() returns with a DirectSound buffer you can use, the function PlayWAVInfo() calls the DirectSound member Play(), whose interface is shown in Figure 5-18. The example code is shown in Listing 5-31.

LISTING 5-31 Calling the DirectSound member Play()

```
LPDIRECTSOUNDBUFFER lpDSB = GetAvailableBuf(lpwi);
if (lpDSB != NULL)
{
  dsval = lpDSB->Play(0,0,dwPlayFlags);
} // if lpwi != NULL
```



The WAVEINFO Driver

You have all the routines you need in DSTOOLS to start playing DirectSound buffers. All the work you spent in developing these functions is now going to pay off in the ease of using these routine calls. For instance, you can make a

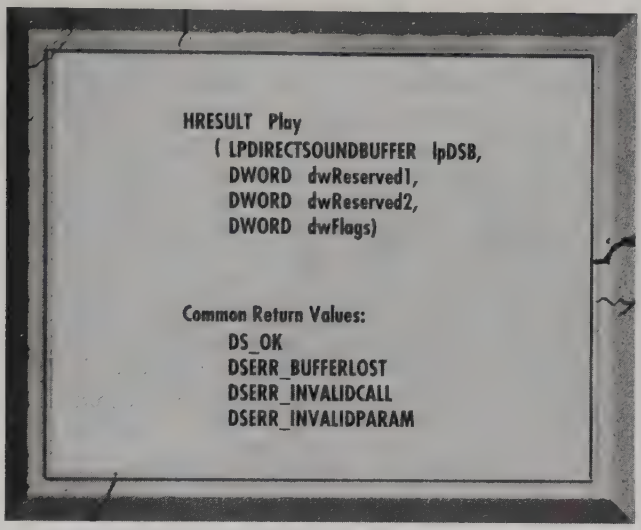


FIGURE 5-18
DirectSound member to play a sound buffer

simple driver to load a resource into the WAVEINFO structure, create a DirectSound buffer, and then play the buffer in a few simple steps. Your driver for DSTOOLS is provided in Listing 5-32.

LISTING 5-32 Simple *Spells of Fury* DSTOOLS driver

```
void DSDriver()
{
  LPWAVINFO lpwi;
  lpwi = CreateWAVInfo(lpDS, "photon", 1);
  PlayWAVInfo(lpwi, DSBPLAY_LOOPING);
}
```

This program can easily be called from the generic game application you put together in Chapter 1. For my code, I put it in the InitGameApp() function for the source file GENX.CPP. The DSDriver() code provides a nice little cut-and-paste code launcher to get you up and running quickly with the DSTOOLS calls. Don't forget to declare your wave resources in the resource script. This is one of my common mistakes I make when coding into the early hours of the morning. The caffeine has long since worn off and my ability to program rapidly diminishes.



A Children's Discovery Game

One of the first multimedia games to be produced was a children's learning adventure called *The Manhole*, on the Macintosh. *The Manhole* was a simple game, coded entirely with Apple's HyperCard scripting language, that allowed

children to explore an *Alice in Wonderland*-style world. Your host was a giant rabbit, who let you into his house to change the channels on his television, go through his drawers, play in his rooms, and explore his subterranean mazes.

The animation in *The Manhole*, and its successor, *Cosmic Osmo*, was very simple. The frames were jerky, and the game had no objective or mission. The idea was for children to look around and satisfy their curiosity by clicking on toon-style devices to see what they would do. Even though the game was intended for children, I know a lot of adults turned their attention away from bug zappers and lava lamps to play with this new-fangled curiosity.

The *Activision* games *The Manhole* and *Cosmic Osmo* engaged the audience's curiosity with whimsical choreographing of animation and sound. The make-believe world was a cross between the *InfoCom Zork* games and a Chuck Jones-directed *Looney Toon*. Children's multimedia games have come a long way since *The Manhole* and *Cosmic Osmo*. Just take a look at the successful Disney line of products, *Lion King*, *Aladdin*, and *Winnie the Pooh*.



Using DSTOOLS in an *Xplore Adventure* Game

You're going to take a look at implementing sound, using your DSTOOLS code, in a children's discovery game, called *Adventure*. There are no photon blasters, warp engines, or demons in *Xplore*. There is just a cast of simple characters that execute an action when their mouse hot spots are double-clicked. Figure 5-19 shows a screenshot of your *Xplore* game. The scene is simple and so is the art.



FIGURE 5-19
Screenshot of *Xplore*

If you haven't already played with the game, fire it up and click on a few things. When the program launches, you see a splash screen and immediately you hear a little jingle playing. This is your DSTOOLS routines playing a WAVEINFO sound. In the file ADV.CPP you can see how you initialized the sounds using DSTOOLS. The code for *Xplore* is shown in Listing 5-33.

LISTING 5-33 Using DSTOOLS in the children's *Xplore* game

```
LPWAVINFO lpwiMusic;
LPWAVINFO lpwiOpenDoor;
LPWAVINFO lpwiKrusty;
LPWAVINFO lpwiTRex;
LPWAVINFO lpwiJump;
LPWAVINFO lpwiLamp;

void InitWAVInfo()
{
  if (lpDS != NULL)
  {
    lpwiKrusty = CreateWAVInfo(lpDS, "krusty", 1);
    lpwiOpenDoor = CreateWAVInfo(lpDS, "OpenDoor", 1);
    lpwiTRex = CreateWAVInfo(lpDS, "TRex", 1);
    lpwiMusic = CreateWAVInfo(lpDS, "Music", 1);
    lpwiJump = CreateWAVInfo(lpDS, "Jump", 1);
    lpwiLamp = CreateWAVInfo(lpDS, "Lamp", 1);
    bSoundEnabled = TRUE;
    PlayWAVInfo(lpwiMusic, DSBPLAY_LOOPING);
  }
}
```

This code resembles the DSDriver() function you wrote previously. The function InitWAVInfo() creates the WAVEINFO sounds after the DirectSound object has been created. This function is called in the game initialization routine (in WinMain()). When the application initializes your sounds, you start playing the theme music for *Xplore*.

In *Xplore*, on the first page of the game, shown in Figure 5-19, there are three hot spots for the user to double-click on. The upper-right side of the cloud is one, the tree is another, and the door to the house is the third hot spot. This kind of reminds me of an interactive children's menu that a restaurant of the future might have. You know, the Baker's Square or Denny's children's menus, where the child looks to see what object doesn't belong in the picture. Oh well, enough of visualization, let's get back to reality.

Click on the door to the house and find yourself standing inside the house, as shown in Figure 5-20. The hot spots are the two windows, the door, the lamp, and the mouse hole. If you were to enhance this code, this would be your portal to a make-believe world. You could script a maze or some rooms to enter if the user selected the mouse hole. Right now the mouse only roars. But play around and see what you think. Show a little curiosity.

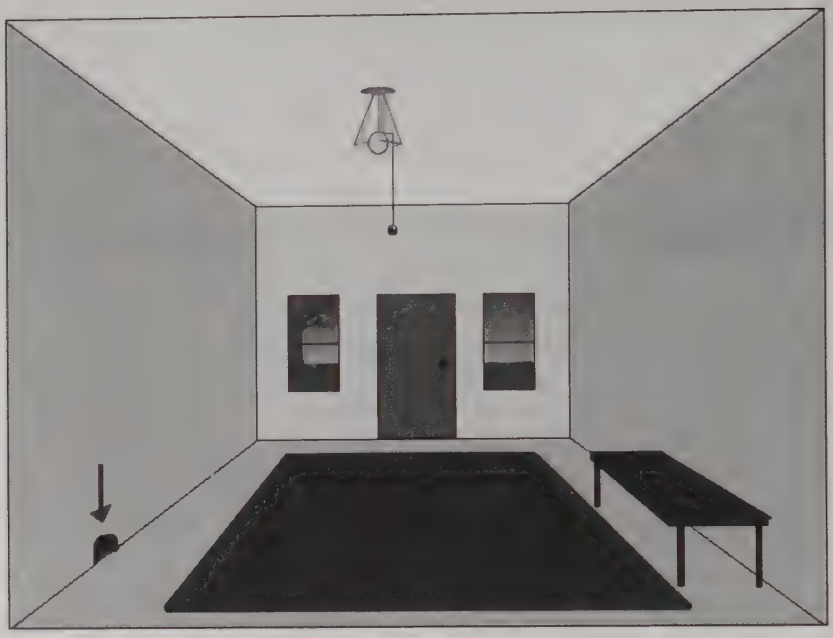


FIGURE 5-20
Interior of the house in *Xplore*

Some General Scripting Required

In *Xplore*, there are no weapon-toting villains, nor does any sprite need to interact with a player action. All the actors in *Xplore* have been given a script and they stick to it. Every act in your script is called a page. When the user sees the first screen of the outside of the house, this is page 1 to your script in *Xplore*. When you move inside the house you are on page 2, and so on.

Each page has an array of functions to handle the scripting involved with each hot spot selected. The first function, such as that used in page 1, is page1Func1() which is used to initialize the DirectDraw surface for the display. The functions after this are pageFunc2() to pageFunc*n*() to handle the various hot spots on this page for the user to select. The code for page script functions can be found in Listing 5-34.

LISTING 5-34 Examples of page scripts used in the *Xplore* game

```
// Go through the front door sequence.
void page1Func1( void )
{
  currPage[4] = '2';
  DDReLoadBitmap(lpDDSOne, currPage);
  newScreen = TRUE;
  pagePtr = page2;
};
```

```
// The sun appears out the clouds for a second.
void page1Func2( void )
{
  Mouse.show = FALSE;
  sequenceExecute(sunClouds);
  Mouse.show = TRUE;
};

// Have something happen in the tree.
void page1Func3( void )
{
  Mouse.show = FALSE;
  sequenceExecute(inTree);
  Mouse.show = TRUE;
};

// Have smoke come out of the chimney.
void page1Func4( void )
{
  Mouse.show = FALSE;
  sequenceExecute(smoke);
  Mouse.show = TRUE;
};
```

Each of these page functions executes a sequence of events when they are activated. A sequence of events is described by a STEP structure, which holds information about blitting RECTs and WAVEINFO sounds to play. The declaration of a STEP structure is shown in Listing 5-35.

LISTING 5-35 The STEP structure used to describe a series of events

```
typedef struct {
  // Source of the graphics area to copy.
  RECT source;
  // Where to put the copied graphics area.
  RECT destination;
  // How long to keep waiting until doing the next step.
  int  time;
  // Sound to generate
  LPWAVINFO *sound;
} STEP;
```

An example of what the STEP data looks like is shown for the STEP *happyFace*. This sequence of STEPs executes when the user double-clicks on the right window from inside the house, which looks like Listing 5-36.

LISTING 5-36 Animating a giggling happy face using the STEP structure

```
STEP happyFace[] = {
//Source              Destination          Time
```

continued on next page

continued from previous page

```
32,  481, 72,  556,  374, 218, 414, 293,  60,   &lpwiKrusty,
73,  481, 113, 556,  374, 218, 414, 293,  60,   NULL,
114, 481, 154, 556,  374, 218, 414, 293,  60,   NULL,
155, 481, 195, 556,  374, 218, 414, 293,  60,   NULL,
196, 481, 236, 556,  374, 218, 414, 293,  180,  NULL,
155, 481, 195, 556,  374, 218, 414, 293,  60,   NULL,
114, 481, 154, 556,  374, 218, 414, 293,  60,   NULL,
73,  481, 113, 556,  374, 218, 414, 293,  60,   NULL,
32,  481, 72,  556,  374, 218, 414, 293,  60,   NULL,
0,   0,   0,   0,    0,   0,   0,   0,    0,    NULL,
};
```

The STEP information is a sequence of source RECTs, destination RECTs, delay time, and WAVINFO sound to be used in this sequence of the *Xplore* game. This STEP is activated by the function sequenceExecute(). This function is a composite of all the DirectX tools you have developed to this point. You are using DDTOOLS for page flipping, some DITOOLS for mouse handling, and DSTOOLS to play a sound. This code is shown in Listing 5-37.

LISTING 5-37 The sequenceExecute() function

```
void sequenceExecute(STEP *sequence)
{
  int     left,top;
  RECT    rcRect;
  DWORD   thisTickCount;

  // Restore the background to a pristine state.
  // Mouse SHOULD NOT be drawn when a sequence is being
  // executed.
  cleanMouse();
  // The sequence always ends with a sequence STEP with a time
  // of zero.  If zero step time is detected then stop sequence.
  while(sequence->time)
  {
    // Get how many ticks this step of the sequence should be
    // on the screen.
    thisTickCount = GetTickCount() + sequence->time;
    // Get top left corner of rectangle that will be copied to.
    left = sequence->destination.left;
    top = sequence->destination.top;
    // Get the source rectangle for the steps cell.
    rcRect = sequence->source;
    // Do the graphics transfer.
    drawOneToBack(left, top, rcRect, TRUE);
    // Detect if any sounds need to be generated
    if (sequence->sound != NULL)
    {
      //playPanned(*sequence->sound);
      PanWAVInfo(*sequence->sound);
    }
```

```
    // Flip the surfaces
    flipSurface();
    cleanMouse();
    // Wait in this loop for certain # of ticks for this step.
    while (GetTickCount() < thisTickCount);
    // Go to the next step in the sequence.
    sequence++;
  }
  // Now we know the sequence is complete. We must make sure that
  // both the Primary, and Secondary surfaces are the same.  Go
  // back to last real step in sequence and put that cell in both
  // surfaces.
  sequence--;
  setBackground(sequence->destination.left,
                sequence->destination.top,
                sequence->source.left, sequence->source.top,
                sequence->source.right, sequence->source.bottom);
}
```

The *Xplore* game is fairly simple in design and you could easily add to it yourself. As I mentioned earlier, you could make another room, or even a maze, branch off from the mouse hole. Whatever you design with this, adventure games such as these are fun for adults and kids. Let the creative juices flow and create interesting little worlds that interact with the user.

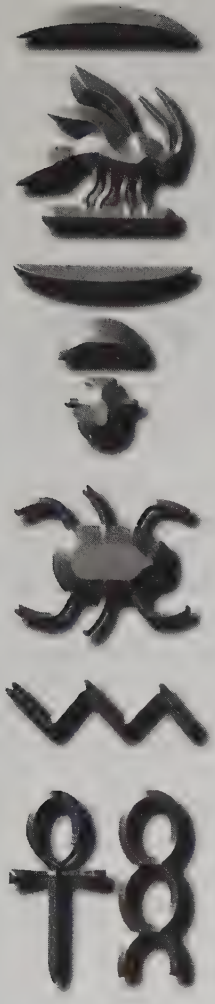


Virtual Mozart

You've covered some pretty lengthy topics on sound files and DirectSound. Once you organized your tools and understood their functionality, you were able to implement sounds to your games to make them more enjoyable. You now have the tools available for you to write your own multimedia operas. Be a virtual Mozart; put tone and feeling into your game's story line. The audio hardware can provide an added sensation in the world you are trying to create for your game. Pull your game player into the game. The mind can be fooled easily. Adhere to the players' senses; tamper with the sensory apparatus they use to determine, in everyday life, what is real and what isn't. Sound is the one sense a computer can actually fool easily. 3D sounds are realistic; 3D graphics are still in their infancy. Sound is the one sense the developer can really hack away at.



TREACHEROUS PLOTS



TREACHEROUS PLOTS

This chapter covers all the basics of scanline conversion algorithms using DirectDraw programming examples. The topics include writing a pixel to a DirectDrawSurface, line-clipping, and simple polygons. These algorithms are frequently used by developers in every aspect of graphics programming for games. Following these topics is a discussion on an object-oriented vector graphics game called *SpaceFoil*. The techniques presented in this chapter will be used to build a simple game that uses polygons and DirectDraw.

EPISODE A DEAD-END



"Oh great, 13, you steered me to a dead-end. Now what?"

"Hold on a minute, Bary," Kala interrupted. "Look at those giant sandstone statues sitting on both sides of this room. I can't make out which Pharaohs they are."

"Hey what's with all the flasks with the purple stuff in it? There's one at the pedestal of each statue," I reported.

"I don't know; why don't you go and pick one up," 13 replied.

"There are a total of six. Three to the right and three on the left. I am going over to the right and pick one up," I commented as I walked over to the first flask. I reached down and picked up the purple, bubbly substance in the jar. At the moment my hand made contact, my mouth involuntarily muttered, "yeah!"

"Yeah, what?" 13 replied from his station.

"I don't know. It just came out when I grabbed the flask," I answered.

"Hey, go grab another one," 13 answered with great enthusiasm.

"I don't think this is such a good idea," Kala added. "I recognize these statues from somewhere. It wasn't in a book though. It was at a digging site in the Valley of the Kings in Egypt."

I walked over to another flask and picked it up. The same thing happened, another "Yeah." I ran around the room and picked up the four remaining flasks.

"Yeah, yeah, yeah, yeah," I yelled with the retrieval of each flask. "I think I'm in a game of some sort."

"Hey, Bary, check out the scroll behind you. It wasn't there before. Go get it," 13 reported.

"Bary, this isn't such a good idea. We don't know where this is leading. Stay put and let me go back and look up these statues in my reference books," Kala pleaded.

"What could possibly happen? It's a game!" I refuted as we walked over to the floating scroll. "This has got to be a map."

I snatched the scroll, and just as I did I heard a rumbling of heavy objects breaking free of their foundations. I didn't want to turn around. I knew what had happened—the statutes had come to life.

"Guys, 13, Kala. I am really open for suggestions at this point in time."

"Run!" Kala and 13 yelled into their microphones.

One of the giant statues put his foot in the path of my escape. In an act of desperation I threw a flask of purple, bubbly liquid at the giant. The substance sizzled away at the sandstone behemoth and dissolved it into a pile of sand.

I threw the remaining flasks at the five other animated statues reducing them to piles of sand.

"Good call, Bary," Kala applauded. "Now get out of there!"

I turned to walk out the door from which I entered, but the hallway I had come into was gone.

"What's the matter, Bary?" 13 asked.

"The game ain't over yet," I answered regretfully.



Plotting Pixels and DirectDraw

The most fundamental operation that can be performed with a DirectDraw object is to write a pixel to a video frame buffer called a DirectDrawSurface. A pixel is a computer acronym for "picture element." A set of pixels is called a bitmap and is used for composing images and textures that can be drawn to the video screen. The Windows 95 Direct X Game SDK DirectDraw object has member functions to blit and stretch images, so why do we even care about pixel plotting? For one thing, it's important to understand the memory organization of a DirectDrawSurface and how to directly manipulate its bits. Another good reason is that texture mapped surfaces, such as the ones used in

DOOM and *Heretic*, require directly writing to offscreen frame buffers to generate the realistic 3D game world. Blitters, such as DirectDraw's BltFast() member function, are typically used to move an entire image or portion of an image from one surface to another. Generating a 3D world involves computing each location of a pixel and writing the pixel directly to the video frame buffer.



Writing to the DirectDrawSurface

The human mind is conditioned to identify a location on a two-dimensional grid as a set of *(x,y)* coordinates. This notion is carried over to computer applications. When you use a paint program, the current cursor coordinates are displayed using *(x,y)* values. Using the Microsoft Direct X Game SDK DirectDrawSurface architecture, video memory is linear. When writing to a linear video frame buffer, such as a DirectDrawSurface, we describe the memory location as having an offset from a pointer. To use *(x,y)* coordinates to describe a two-dimensional space we must use mappings. A mapping is a translation from *(x,y)* display coordinates to a physical offset in memory of the pixels' location in a DirectDrawSurface.

We performed a similar operation back in Chapter 2 when we worked with DIBs. But as you remember, DIBs were confusing to work with because their orientation was bottom-up. A DirectDrawSurface is a little easier to work with because its orientation is top-down. If you're unfamiliar with the bottom-up and top-down bitmap organization, go ahead and take the time to refresh yourself with the discussion in Chapter 2.



Display Coordinates to DirectDrawSurface Translations

In order to write a pixel intensity to a DirectDrawSurface you must know the physical attributes of the buffer, such as the surface height and width. When you create a surface you use the DirectDrawSurface descriptor fields to tell DirectDraw the size of surface you need to allocate. For example, if you have a 640 x 480 pixel DIB in system memory and you want to copy into a DirectDrawSurface, you ask DirectDraw to allocate a 640 x 480 offscreen surface and you assign it the pointer, *lpDDSOffscreen*, which references the first pixel in the DirectDrawSurface. Every pixel thereafter is located at a linear offset from this location. Now the offscreen buffer, the DirectDrawSurface in video memory, is linear (one-dimensional), but we describe all of our display attributes with two-dimensional *(x,y)* coordinates. You need a mapping from display *(x,y)* coordinates to DirectDrawSurface memory locations. Even though the DirectDrawSurface is one-dimensional, you use two-dimensional attributes to create the buffer. Figure 6-1 shows a 640 x 480 pixel resolution display and its video frame buffer. The video frame buffer is laid out so that the

first 640 pixels comprising the first row on the display are the first 640 locations in the frame buffer. The next 640 locations in the buffer comprise the second row of scanlines for the display. Consecutively laying out 480 rows of 640 pixels is how our example DirectDrawSurface is composed in video memory.

This memory scheme allows you to use a simple relationship to translate display *(x,y)* coordinates into a DirectDrawSurface location. Any pixels offset into a DirectDrawSurface can be calculated with the following relationship:

Pixel(x,y) = *lpDDSOffscreen* + *y*(Xmax)* + *x*

The pointer *lpDDSOffscreen* points to the first pixel in the DirectDrawSurface. Rows of pixels are stored consecutively one after the other. To locate the desired row offset into the DirectDrawSurface, make the calculation *y*Xmax*. Using the 640 x 480 pixel DirectDrawSurface, *Xmax* is equal to 639. Remember, screen coordinates start at 0 and each row contains pixels 0 through 639 for a total of 640 pixels. Once you have located the start of your row in the buffer, add the offset *x*, which is the pixel location in the DirectDrawSurface you were after.

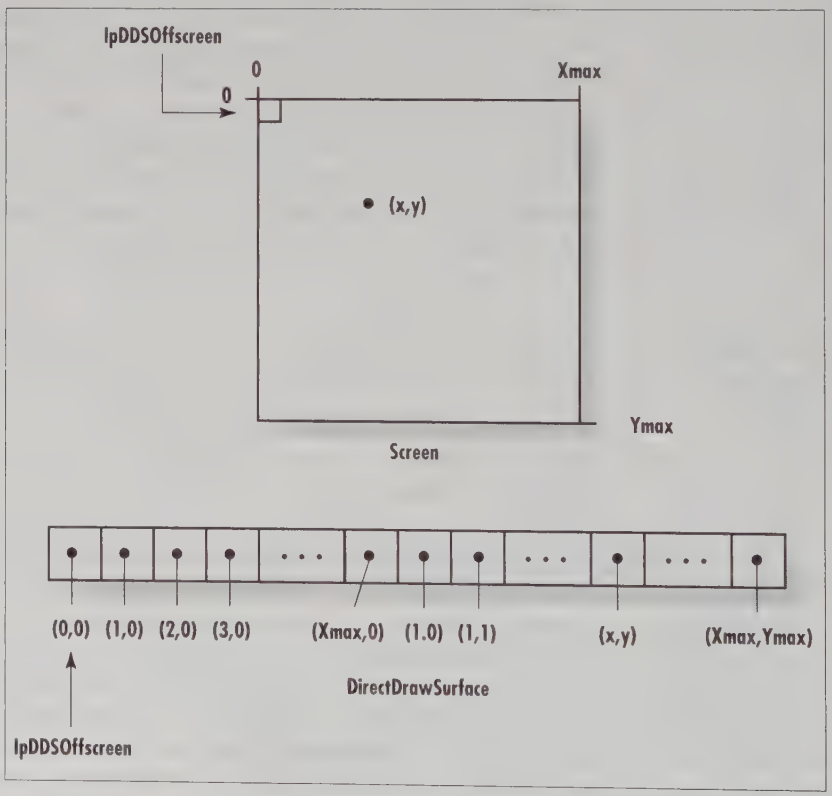


FIGURE 6-1
Display coordinates and DirectDrawSurface architecture

Getting a Pointer to a DirectDrawSurface

In the DOS environment you would normally calculate your pixel-to-offset translation and write your pixel to that location in the frame buffer. In the Windows 95 environment your applications must adhere to multitasking coding conventions. You can't simply write to a buffer at will, because another application or a background task may be writing to the same buffer. You must request access to a DirectDrawSurface by calling the DirectDrawSurface member function Lock(). Figure 6-2 shows this member function. If the call to Lock() is successful, it will return a pointer to the locked surface in a DDSURFACEDESC structure. Your application will request access to the surface by calling Lock() and providing a RECT structure of the bounding area on the surface you want to access. Your application can request access to the entire surface by calling Lock() with a NULL RECT pointer. This will give your application exclusive access to the surface.

The member function Lock() essentially locks the surface area described by your RECT structure and returns a pointer to that location in the surface. Because the memory is locked, you need to write your pixel(s) and then release the locked memory for other tasks to use. When your application or background task is through writing to the DirectDrawSurface it needs to call the Unlock() member function, as shown in Figure 6-3. Every Lock() call requires an Unlock() call by your application. Your application may Lock() the same surface multiple times. A DDSURFACEDESC structure is used to determine which Lock() and Unlock() are associated.

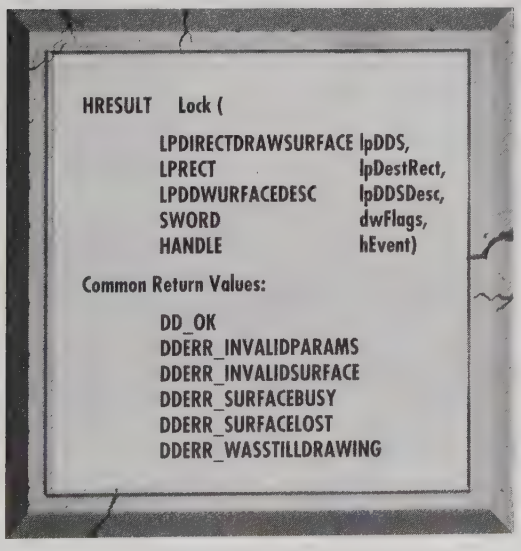


FIGURE 6-2
Accessing a DirectDrawSurface

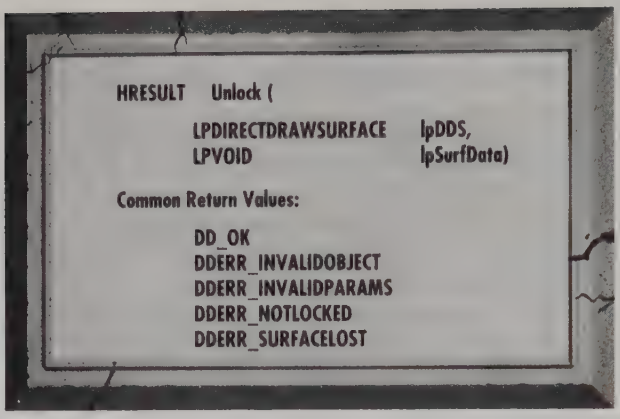


FIGURE 6-3
Unlocking a DirectDrawSurface after a write



Writing a Pixel Using the DDTOOLS Code

In the *Spells of Fury* canned DirectDraw routines that have been put together, called DDTOOLS, there is a function for writing a pixel to a DirectDrawSurface. The DDTOOLS routine writeSurface() is what you'll use for the pixel plotting examples in this book. The function is presented in Listing 6-1.

LISTING 6-1 The DDTOOLS function writeSurface()

```
void writeSurface( LPDIRECTDRAWSURFACE lpDDS,
                   UINT x,
                   UINT y,
                   BYTE colorValue )
{
  UINT bitsIndex;
  DDSURFACEDESC DDSDesc;
  LPSTR lpDstBits;
  HRESULT ddrval;

  DDSDesc.dwSize = sizeof(DDSDesc);
  while(1)
  {
    ddrval = lpDDSBack->Lock(NULL, &DDSDesc, 0, NULL);
    if(ddrval == DD_OK)
      break;
    if(ddrval != DDERR_WASSTILLDRAWING)
      return;
  }
  lpDstBits = (LPSTR) DDSDesc.lpSurface;
  bitsIndex = 640 * y + x;
  lpDstBits[bitsIndex] = colorValue;
  lpDDSBack->Unlock(NULL);
}
```

The writeSurface() function requires four parameters to write a pixel to a surface. The first parameter, LPDIRECTDRAWSURFACE *lpDDS*, is the pointer to the surface you want to write to. The next two parameters are the (x,y) display coordinates to write to in the surface. The fourth parameter to use is the pixel intensity (color). In the body of the function, declare an offset into the surface, *bitsIndex*, a DDSURFACEDESC *DDSDesc*, and a pointer to the locked surface, *lpDstBits*.

The code used to lock the surface is presented in Listing 6-2.

LISTING 6-2 Locking the back flipping surface

```
while(1)
{
  ddrval = lpDDSBack->Lock(NULL, &DDSDesc, 0, NULL);
  if(ddrval == DD_OK)
    break;
  if(ddrval != DDERR_WASSTILLDRAWING)
    return;
}
```

You need to use a loop to lock the surface because other tasks or applications may be using the surface you requested. Note that you passed a NULL RECT structure pointer as you were requesting to lock the entire surface. You may want to make this function more robust. The DirectDrawSurface member BltFast() could be copying a portion of the surface to a flipping surface while you're executing the call to Lock(). This while loop essentially provides a mechanism to avoid surface write collisions from one or more tasks. When you successfully lock the surface, you want the loop to break. If the Lock() fails, you abort the attempt to write a pixel.

When the DirectDrawSurface member Lock() returns successfully, it fills in your DDSURFACEDESC structure with the information you need to access the locked surface. You can now assign your pointer to your locked surface, like this:

```
lpDstBits = (LPSTR) DDSDesc.lpSurface;
```

The pointer, *lpDstBits*, is pointing to the first pixel in the offscreen surface. You need to calculate the offset into the surface of the location you wish to write to. The code used in the DDTOOLS routine assumes a surface width of 640 pixels. This code looks like this:

```
bitsIndex = 640 * y + x;
```

And finally, you can write your pixel and release the locked memory with:

```
lpDstBits[bitsIndex] = colorValue;
lpDDSBack->Unlock(NULL);
```



Writing Pixels in Review

Drawing pixels is one of the most primitive graphics functionalities used by even the most complex drawing routines. Pixels are described as *(x,y)* pair screen coordinates and are translated to respective frame buffer memory locations. A frame buffer for the Windows 95 Direct X Game SDK is called a DirectDraw surface.

Writing to a specified location on a DirectDraw surface requires calling the DirectDraw member function Lock(). The call to Lock(), if successful, returns a special pointer called a surface descriptor, which allows you to manipulate the bits at a specified location. Writing to the surface is achieved by initializing a pointer to reference the descriptor's surface. Multiple tasks may be attempting to compose a single surface, so when a task finishes rendering, it must Unlock() the specified surface location.



Drawing Lines

Today's game market is focused on producing 3D texture-mapped games. The topic of the last section was writing pixels, which is useful information for texture mapping. But why in this day and age do we want to know how to draw lines? Line drawing routines are useful in a variety of game applications. Texture-mapped games use line-drawing algorithms to display maze maps. Older line-drawn (vector graphics) games such as *Asteroids*, *Asteroids Deluxe*, *Major Havoc*, *Missile Command*, *Tail Gunner*, *Lunar Lander*, *Tempest*, and *Star Wars* are being re-introduced to an audience that was too young to play them the first time around. Yes, folks, there was a *Star Wars* game released in the early 1980s that was completely vector graphics. I recently saw one of these old coin-op machines still in use at Disneyland inside a StarTours gift shop. I was amazed at how many quarters kids were dumping into a game that didn't have any texture-mapped ninja warriors. Another old game finding new life is the Atari game*Tempest*, which was just rereleased for the Jaguar game console. Most high-end game developers have focused too much on multimedia and not enough on the fun and entertaining aspect of gameplay. One game comes to mind where I literally went out and bought a serious joystick to play a 3D-rendered ninja-fighting robot game. It turned out the ultra-cool looking game was just as bad with the super joystick. I ended up breaking in the joystick with a few rounds of Atari's old vector graphics *BattleZone* tank game. *Write a cool game and they will come.*

Two Points Describe a Line

A line segment can be described by a set of two (x,y) coordinates called endpoints, as shown in Figure 6-4. The endpoint P_1 has coordinates (x_1, y_1) and the second endpoint P_2 has coordinates (x_2,y_2). For humans, this is a simple task of getting out a ruler and connecting the two dots. For a computer, the task isn't so trivial, especially for the microcomputer, which relies on software to determine how to draw a line. In the next few years, microcomputers will probably have the hardware that SGI and UNIX workstations have to draw lines and not require software to do this. In the meantime, however, we're stuck with providing line drawing functionality to our software.

The Slope of a Line

The basic functionality you need to provide with a line drawing routine is to plot pixels in a straight path from one endpoint to another. The position of each pixel is computed from an equation describing the line. The equation for a line is given in Equation 6-1.

Equation 6-1 Slope-intercept equation for a line

$$y = mx + b$$

The variable m is called the slope of the line and the variable b is the y-intercept. The y inter-what? Hold on, we'll laboriously go through the nomenclature. If you're chuckling to yourself because you know what I'm talking

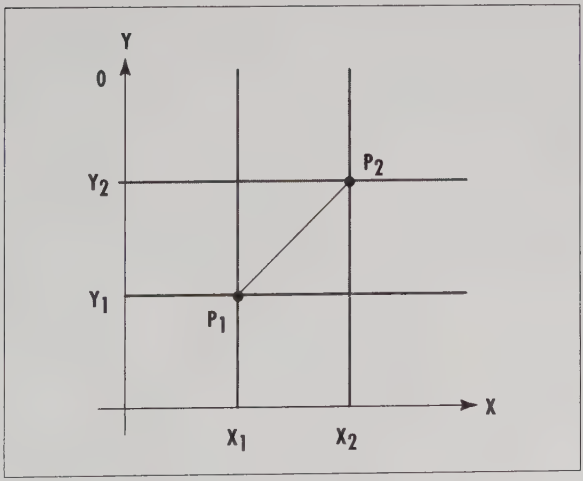


FIGURE 6-4

A line described by coordinates (x_1, y_1) and (x_2, y_2)

about, just skip ahead and look at the pretty figures. We'll catch up with you in the discussion of the Bresenham line-drawing algorithm.

Earlier, a line segment was described as having two endpoints, $P_1(x_1, y_1)$ and $P_2(x_2,y_2)$. These four values can provide us with critical information about the line. For instance, is it a vertical line, is it a horizontal line, does it go off at a tangent, and if so what direction? Wow, that's a good deal of information from two coordinates. How do you get all this information from just two points? Simple. The variable *m* from the line equation is called the slope of the line. The slope is a ratio representing the change in the *y* endpoints with respect to the change in the *x* endpoints. The slope *m* of a line is calculated using Equation 6-2.

Equation 6-2 Computing delta *x*, delta *y*, and the slope of a line

run: $dx = x_2 - x_1$
rise: $dy = y_2 - y_1$
slope: $m = dy / dx$

The variables delta *x*, *dx* (pronounced dee-x), and delta *y*, *dy* (pronounced dee-y), tell you about the rise, *dy*, and the run, *dx*, of the line. The slope of the line, *m*, tells you how steeply the line rises or falls, as shown in Figure 6-5. For example, given the first point P_1 with coordinates (2,1) and a second point P_2 with coordinates (4,5), we can compute the slope of the line.

$$m = dy/dx = (y_2 - y_1)/(x_2 - x_1) = (5 - 1)/(4 - 2) = 4/2 = 2$$

In layman's terms, a slope of $m = 2$, means that *y* increases two units every time *x* increases by one. The y coordinate changes two times faster than the *x* coordinate does. From a computer graphics perspective, for any *x* interval, *dx*, along a line, we can calculate the corresponding *y* interval, *dy*. For you

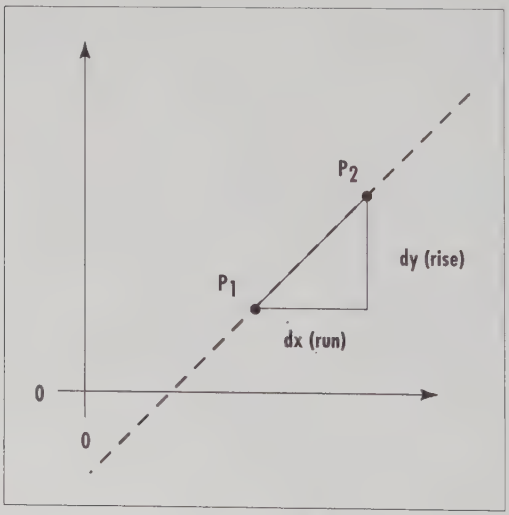


FIGURE 6-5
The slope of a line



old-timers out there, you'll appreciate the fact that my old text describes setting the analog device's voltage proportional to *dx* to get the desired horizontal deflection and change the vertical deflection voltage proportional to *dy*. But enough reminiscing, let's take a look at Equation 6-3, the Point-Slope equation.

Equation 6-3 The Point-Slope equation

$dy = m * dx$

This states that as long as you know the slope, *m*, of a line, and you know either *dx* or *dy*, you can calculate the corresponding *dy* or *dx*. This simple little equation is used a lot in game design. This computation is intensively used in raycasting, texture mapping, and for drawing polygons.



The *y*-Intercept

The ideal world, and extremely boring video games, would use perfectly vertical lines where the points on the line all had the same *x* values, $x = x1 = x2$, as shown in Figure 6-6. Line drawing in games is rarely this simple. Typically, lines are diagonal, pointing in a direction based on the user's input. These lines are usually nonvertical or diagonal.

Lines that are not vertical must cross the *y*-axis at some point $(0, b)$. The variable *b* is called the *y*-intercept of the line. A *y*-intercept is the *y* coordinate where the line crossed the *y*-axis. A diagonal line with its *y*-intercept is shown

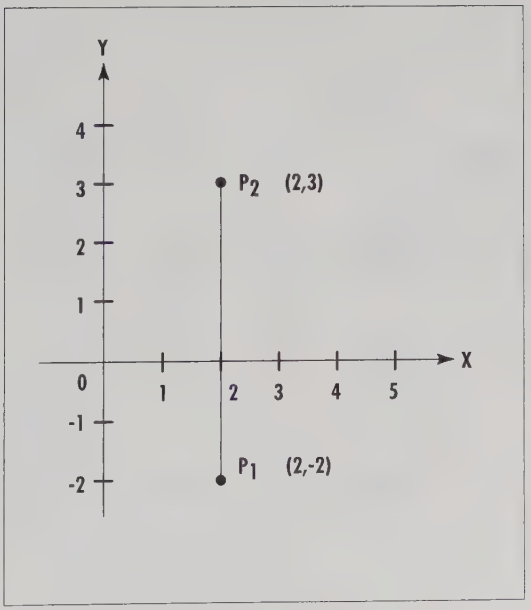


FIGURE 6-6

$x = 2$ forms a vertical line

in Figure 6-7. If the endpoint P_1 is your *y*-intercept and endpoint P_2 is (x_2,y_2), the Point-Slope equation looks like this:

```
y2 - y1 = m * (x2 - x1)
y2 - b = m * (x2 - 0)
      y2 = mx1 + b
```

The equation $y = mx + b$ is your Point-Slope equation! This handy equation will allow you to calculate *y* values for your second endpoint, just by knowing the *x* value of your current endpoint and the *y*-intercept of the line. It's a very handy tool. If you ever do any Wolfenstein-style raycasting, the Point-Slope equation is your friend. Other useful line equations are provided in Table 6-1.

TABLE 6-1 Useful line equations

LINE	EQUATIONS
x = a	Vertical line through (a,b)
y = b	Horizontal line through (a,b)
y = mx+b	Slope-intercept equation
y2-y1 = m(x2-x1)	Point-slope equation

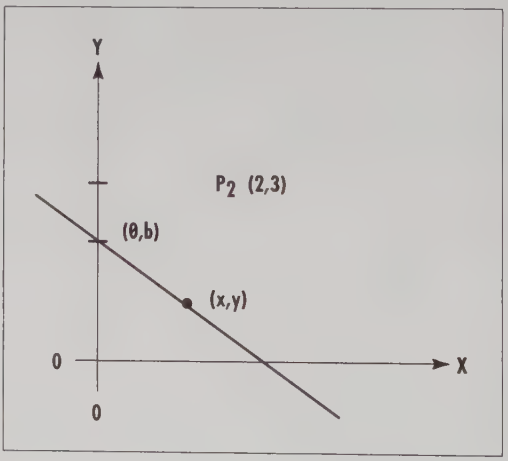


FIGURE 6-7
The $y = mx + b$ line



The Bresenham Scanline Drawing Algorithm

In the early 1960s, a scientist by the name of Bresenham wrote an algorithm to efficiently determine pixel positions on a line using only integer arithmetic. History reflects that Bresenham was working on a plotter design for IBM, but we all know he was really working on a Lunar Lander video game.The Bresenham algorithm works by generating an approximation of a line on the display. The pixel spacings provide a finite resolution; therefore, portions of the true line may not accurately fall precisely on a pixel, as shown in Figure 6-8.

As you can see in the figure, the line either falls directly on a pixel, or is directly above or below a pixel. The Bresenham algorithm draws the pixel if it lies directly in the path of the true line. Otherwise, the algorithm provides a mechanism to check deviation of current and adjacent pixels close to the true line. The deviation is classically now regarded as the error term. The algorithm works by progressing along the line, one pixel at a time. If the pixel doesn't lie directly on the true line, the error term is checked to see which pixel is a suitable approximation to the true line.

So, how do you know which axis to proceed along to draw pixels? You must make a decision in your program by looking at the *dx* and *dy* values in the Point-Slope equation. For example, let's plot a simple line on the screen from (0,0) to (3,1). Plugging the values into the Point-Slope equation you have

```
m =  (y2 - y1)/( x2 - x1) =  (1 - 0)/( 3 - 0) = 1/3
```

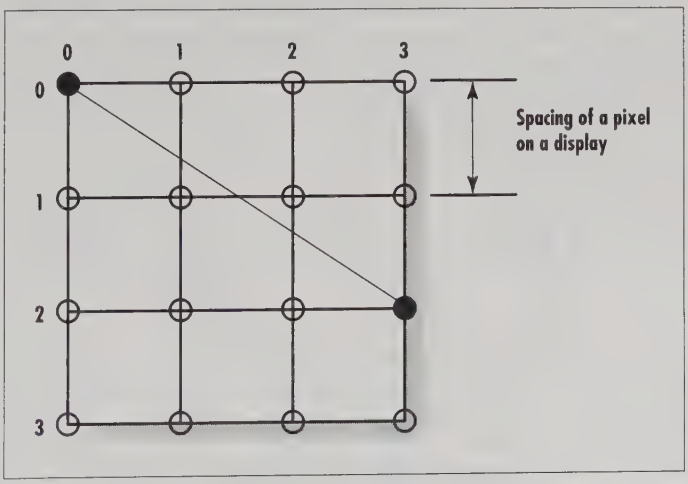


FIGURE 6-8
Line and finite pixel spacing on a display

The Point-Slope equation says the *y* axis is only incremented by one unit for every three-unit increment of the *x* axis. The Bresenham algorithm selects the *x*-axis as its primary axis and the *y*-axis as its approximation axis. This was determined because the run, *dx*, is greater than the rise, *dy*. That is, the *x* coordinate is changing faster than the *y* coordinate. In this particular case, the approximation of the line is made by stepping one pixel unit across the primary axis each time a pixel is drawn, and moving one pixel unit in the approximation axis direction each time the error term for the line is more than halfway between pixels on the approximation axis. In Figure 6-9, you can see that the four values for the primary axis, *x*, will be 0, 1, 2, and 3. It is the values on the approximation axis, *y*, that you will have to determine.

As an example, walk through plotting a line for the endpoints (0,0) and (3,1). The initial point to plot is easy. It's (0,0) because it is an endpoint. The next point to plot is not so trivial; the *x* coordinate is already known to be one. But what's the *y* value going to be? This is where the error term comes into play and checks the true line deviation, as shown in Figure 6-10. In the figure the error term reveals the true line is actually less than halfway between the current *y* pixel and the adjacent *y* pixel, so the algorithm chooses the current *y* pixel and plots the point (1,0).

The algorithm states that when a pixel is drawn, the primary axis coordinate (the *x* coordinate in this case) is incremented one pixel unit and the next *y* pixel coordinate is approximated. Figure 6-11 shows that in plotting this location, you need to select the adjacent *y* pixel because the error term from the true line deviates more than halfway between the two *y* pixels. The new point the algorithm is going to draw is (2,1). When a pixel is drawn it increments the primary axis by one unit and proceeds to evaluate the final coordinate. You already know that this is an endpoint, but notice that the algorithm went to the precise *x* coordinate and the *y* coordinate, which is the correct endpoint coordinate. The algorithm determines this location to have an error term of zero and proceeds to draw the pixel.

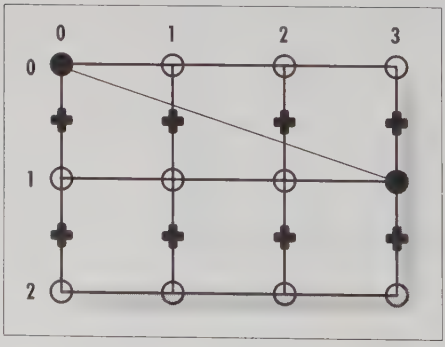


FIGURE 6-9
Bresenham's algorithm approximating *y* values

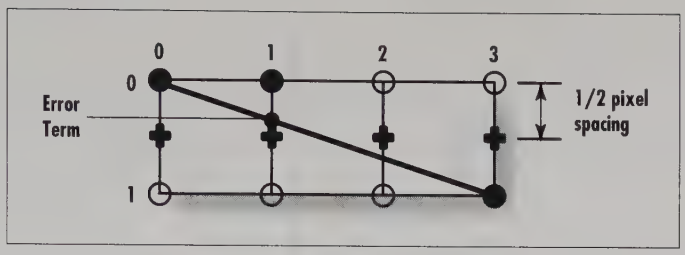


FIGURE 6-10

Plot current *y* pixel, error term < 1/2



Distinguishing the Primary Axis and Its Slope

The key to implementing the Bresenham line-drawing algorithm is to distinguish which axis is the primary axis and which is the approximation axis. The top-down approach shows that the origin of the screen (0,0) is at the upper-left-hand corner of the screen. The *y* axis increases in the downward direction and the *x* axis increases to the right. You can visualize a logical coordinate system made up of four quadrants. Quadrant I is the actual screen coordinates themselves and quadrants II, III, and IV are logically bound to the screen to describe possible directions along the axes, as shown in Figure 6-12.

It is obvious from the diagram that quadrants II through IV are logical; they don't physically exist. Why are they needed? In order to decide which axis to choose as the primary axis, you need to examine the orientation of the line. Orientation means is *x* or *y* increasing or decreasing from the starting endpoint to the final endpoint?

The orientation of a line can be checked by looking at its slope. Figure 6-13 shows what happens when you subdivide the four quadrants by drawing two lines directly through the origin. These two lines are $y = x$, which subdivides quadrants Q1 and QIII, and the line $y = -x$, which subdivides quadrants QII and QIV. By subdividing the quadrants with these two lines some important relationships about the behavior of the slopes of a line are revealed. But this also shows that for every quadrant there are two possible line orientations, making a total of eight possible conditions the Bresenham line-drawing algorithm must handle.

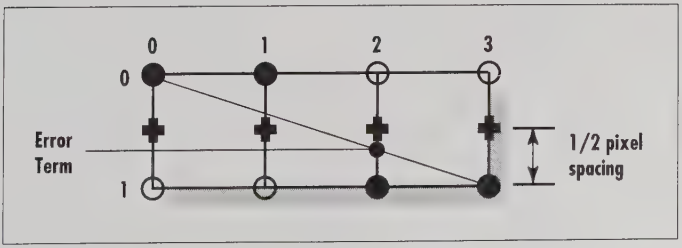


FIGURE 6-11

Plot adjacent *y* pixel, error term > 1/2

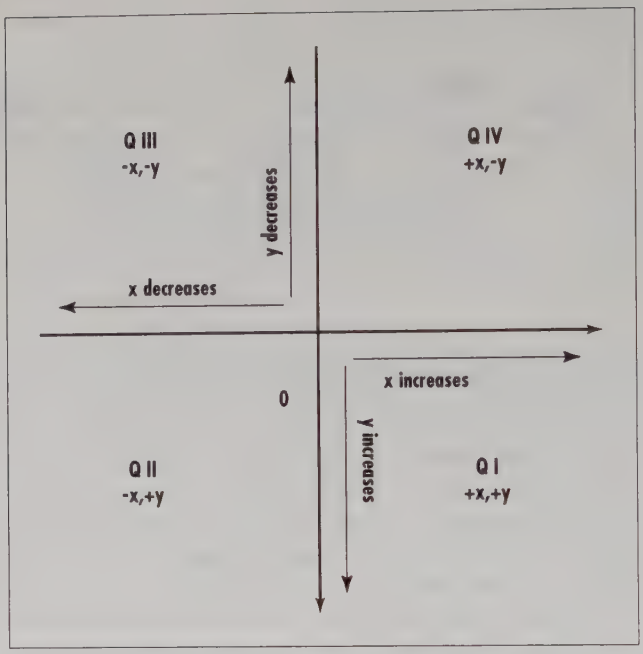


FIGURE 6-12
Four quadrants indicate direction along the axes

A nifty little trick illustrated by Michael Abrash in his book, *The Zen of Graphics Programming*, is to swap the P1 and P2 endpoint coordinates for every line where $dy < 0$. So if the *y1* coordinate of endpoint P1 is greater than the *y2* coordinate of endpoint P2, swap the coordinates of P1 and P2. The result of this operation diminishes the number of cases the algorithm has to deal with (see Figure 6-14). After all, the line hasn't been affected; two endpoints determine a line, not the order of the endpoints.

The number of cases of line orientation in the Bresenham line-drawing algorithm is now reduced to four possibilities. These cases include the two criteria for which *x* is the primary axis and the two criteria for which *y* is the primary axis. Figure 6-14 shows that if the absolute value of *dx*, | *dx* | is greater than *dy*, there are actually two cases that are similar and only differ in the direction of the *x* itself. Likewise, if the absolute value of | *dx* | is less than *dy*, the two cases are similar and differ only in the direction of the *y*. So, now the four cases can be reduced to only two, as shown in Figure 6-15.

What has all this got to do with Bresenham's line-drawing algorithm? The earlier example approximated the next *y* pixel coordinate based on whether it was more than halfway to the adjacent *y* pixel. The two cases just isolated tell which axis is the primary axis and which is the approximation axis to apply the algorithm. If the absolute value of *dx*, | *dx* |, is greater than *dy*, then the *x* axis becomes the primary axis and the *y* axis is the approximation axis. If the absolute value of *dx*, | *dx* |, is less than *dy*, the *y* axis is the primary axis and the *x* axis is the approximation axis.

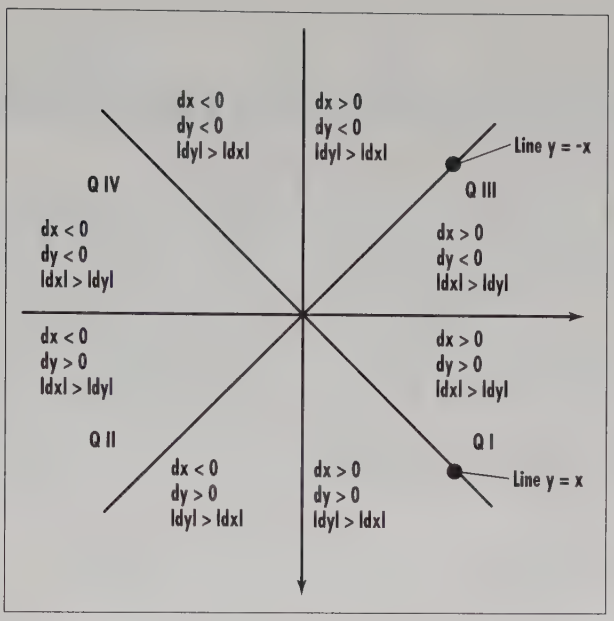


FIGURE 6-13

Orientation of a line in each of the four quadrants

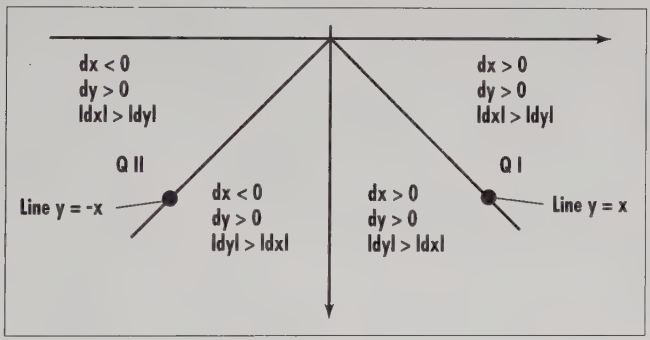


FIGURE 6-14

Results of swapping endpoints for $dy < 0$

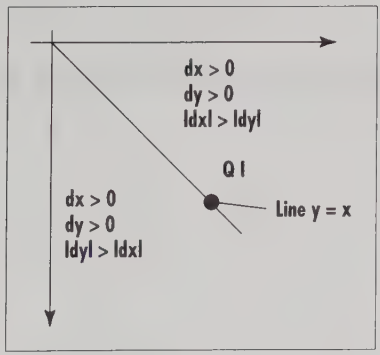


FIGURE 6-15

Two cases for line orientation



Implementing Bresenham's Line-Drawing Algorithm

Enough with Linear Math 101, already. Now to get down to business and do some code. Earlier, the eight cases of line orientation the algorithm had to handle were defined. Thanks to a little trick provided by Michael Abrash, the number of cases could be cut in half by swapping endpoints, if necessary. The very first thing to do in the code is perform this coordinate swap if it is necessary. The code to check and perform the swap is shown in Listing 6-3.

LISTING 6-3 Swapping *y* terms

```
if (y1 > y2)
{
  swapTermsInt(&y1, &y2);
  swapTermsInt(&x1, &x2);
}
```

The function swapTermsInt() is a simple utility included with the *Spells of Fury* code. Its operation is trivial. If you're curious about it, just flip to the full listing of the Bresenham Line-Drawing code provided in this section.

The next operation that needs to be dealt with is the computation for the slope. You don't actually calculate the slope, because you want to maintain integer arithmetic and multiplication for speed. You will rely solely on the delta values *dx* and *dy*.

```
int dx, dy;
dx = x2 - x1;
dy = y2 - y1;
```

To further reduce the number of cases to handle, we made a decision to determine if the *x* direction of the line was increasing or decreasing, see Listing 6-4.

LISTING 6-4 Checking the direction of a horizontal line

```
if (dx > 0)   // is direction of x increasing?
{
  step = 1;
}
else  // no, x is decreasing
{
  dx = -dx;
  step = -1;
}
```

No, I didn't get cheesy on you guys like some college texts do. You know, the unforgettable, *the following solution is left as an exercise for the reader*. The actions taking place in this portion of code are easier to see if the code within the conditional statements is omitted.

Next fill in the blanks for the rest of the code. Fortunately, the code to fill in is under the comment, *handle two cases* and is identical, regardless if *x* is increasing or decreasing. Then perform a check against *dx* to see if it's greater than *dy*. If this is TRUE, then identify the *x* axis as the primary axis and the *y* axis as the approximation axis. If *dx* is greater than zero, from the initial condition, we use the code in Listing 6-5.

LISTING 6-5 The case for $dx > 0$

```
step = 1; // proceed in the positive direction
if (dx > dy)
  ApproxNextY( x1, y1, dx, dy, step, color );
else
  ApproxNextX( x1, y1, dx, dy, step, color );
```

If *dx* is less than zero, use the else statement from Listing 6-6.

LISTING 6-6 The case for $dx < 0$

```
step = -1; // proceed in the negative direction
if (dx > dy)
  ApproxNextY( x1, y1, dx, dy, step, color );
else
  ApproxNextX( x1, y1, dx, dy, step, color );
```

The code is nearly identical except for the step value which is either positive or negative. The pixel step value is dependent on the sign of *dx*. The purpose for splitting up the two cases in the conditional statements is to set the proper step value. The code used for this is shown in Listing 6-7.

LISTING 6-7 Setting the *step* value

```
if (dx > 0)
{
  step = 1; // proceed in the positive direction
}
else
{
  dx = -dx;
  step = -1;  // proceed in the negative direction
}
if (dx > dy)
  ApproxNextY( x1, y1, dx, dy, step, color );
else
  ApproxNextX( x1, y1, dx, dy, step, color );
```

This completes the code for the Bresenham line-drawing algorithm to make decisions on which is the primary axis and which is the approximation axis. To complete the code provide two functions, one that approximates the correct *y* pixel when *x* is the primary axis, and the other that approximates the correct *x* pixel when *y* is the primary axis. Name these functions ApproxNextY(), and ApproxNextX(), respectively.



Approximating the *y* Pixel

The function ApproxNextY() will handle drawing lines where $dx > dy$. In this case, the primary axis is *x*. The *x* pixel will be plotted successively in the $x + 1$ direction or the $x - 1$ direction depending on the step value (+1 or -1). Now calculate three constants, denoted *k1*, *k2*, and *errorTerm*.

```
k1 = dy * 2
k2 = 2 * (dy - dx)
errorTerm = k1 - dx
```

For the line, P1(0,0) to P2(3,1) use,

```
dy = y2 - y1 = 1 - 0 = 1
dx = x2 - x1 = 3 - 0 = 3

k1 = dy * 2 = 2
k2 = 2 * (dy - dx) = -4
errorTerm = k1 - dx = 2 - 3 = -1
```

Using these initial conditions, the function will immediately plot the point P1(0,0). From here the *x* axis is bounded by our points x_1 and x_2. Enter a loop and walk the path of length *dx*. Hence, the loop will iterate *dx* times with each iteration decrementing the value of *dx*. Using the previous example of the line P1(0,0) and P2(3,1), the following initial conditions exist when the loop is entered.

```
loop: dx times
                                        dx = 3      errorTerm = -1
x = 0          y = 0
```

Graphically, we have plotted the first point, by default, as shown in Figure 6-16.

Plotting the first endpoint is trivial. The points hereafter require careful inspection and handling of the *errorTerm*. The *dx* points will be plotted successively. However, to approximate the *y* coordinate, examine the value of the *errorTerm*. If it is less than zero merely adjust the *errorTerm* by adding the constant *k1*. Then increment the *x* coordinate and plot the pixel using the current *y* value. Listing 6-8 demonstrates the updating of the *errorTerm*.

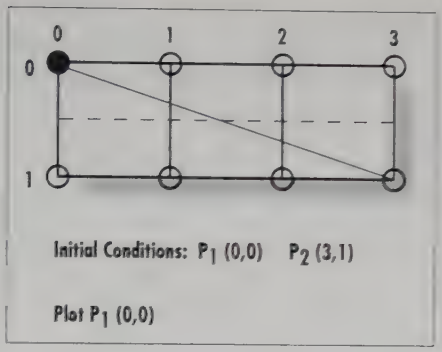


FIGURE 6-16
Initial conditions for function ApproxNextY()

LISTING 6-8 Update the *errorTerm*

```
if (errorTerm < 0)
  errorTerm += k1;
x += step;
writePixel(x,y);
```

If the *errorTerm* is greater than or equal to zero, Listing 6-9 shows how we update the *y* coordinate.

LISTING 6-9 Update the *y* coordinate

```
if (errorTerm >= 0)
{
  errorTerm += k2;
  y++;
}
x += step;
writePixel(x,y);
```

These two conditional statements are contained within a loop that iterates *dx* times. During each iteration of the loop check to see if there is a need to increment *y* to the adjacent pixel or keep the current *y* value. Using the initial conditions from the example results in:

```
dy =1   dx = 3    k1 = 2   k2 = -4   errorTerm = -1
```

Entering the first iteration of the loop,

```
loop = 0    dx = 3   x = 0   y = 0   errorTerm = -1
```

The conditional statement tests the *errorTerm* and determines that it is less than zero. The following operations take place on *dx*, *x*, *y*, and the *errorTerm*:

```
errorTerm < 0
    errorTerm = errorTerm + k1 = -1 + 2 = 1                    errorTerm = 1
x = x + 1
x = 1
plot(x,y)
x = 1 y = 0
decrement dx
dx = 2
```

The result of first iteration of the loop is shown in Figure 6-17. Next proceed into the next iteration of the loop and determine where to plot the next pixel. The last iteration of the loop has the following values,

```
loop = 1  dx = 2  x = 1  y = 0  errorTerm = 1

errorTerm > 0
  errorTerm = errorTerm + k2 = 1 + (-4) = - 3                    errorTerm = -3
  y = y + 1
y = 1
x = x + 1
x = 2
plot(x,y)
x = 2 y = 1
decrement dx
dx = 1
```

In this iteration, the decision was made to advance the y coordinate. Figure 6-18 shows how the line is beginning to form. One final loop is required to complete the example. It's interesting that even though the final coordinate is known, $P_2(3,1)$, (see Figure 6-19), the Bresenham line-drawing algorithm will approximate it, and do so quite accurately. From the previous loop, we have the following conditions,

```
loop = 0  dx = 1  x = 2  y = 1  errorTerm = -3

errorTerm < 0
  errorTerm = errorTerm + k1 = -3 + 2 = -1                       errorTerm = -1
x = x + 1
x = 3
plot(x,y)
x = 3 y =1
decrement dx
dx = 0
```

You have now walked through the plotting of a line using the Bresenham Line-Drawing algorithm for a line where the value of *dx*, the width of the line, is greater than *dy*, the height of the line. With this sound understanding you can take a look at the code for ApproxNextY(), and at the same time get that ethereal feeling of harmony, and the balance of what the code does. The result of the

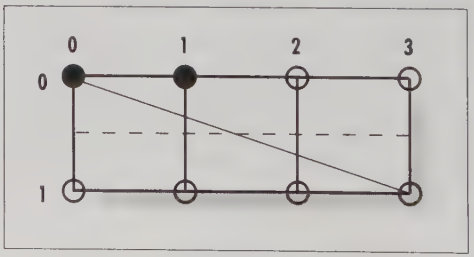


FIGURE 6-17
Plotting the first pixel

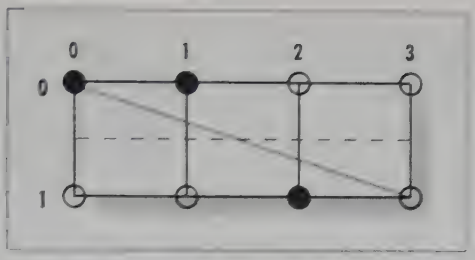


FIGURE 6-18
Incrementing the *y* Coordinate

code is shown in a screenshot in Figure 6-20. It's that same euphoric feeling you get when you listen to Axel Rose play his *Estranged Rock Trilogy* in D-Flat. With no further ado, I present the function, ApproxNextY(), in Listing 6-10.

LISTING 6-10 The ApproxNextY() function

```
void ApproxNextY( UINT xp, UINT yp,
                  UINT dx, UINT dy,
                  int step, BYTE color )
{
  int k1, k2, errorTerm;

  k1 = dy << 2;
  k2 = k1 - (int)(dx << 2);
  errorTerm = k1 - (int) dx;
  writePixel(xp, yp, color);
  while (dx--)
  {
    if (errorTerm >= 0)
    {
      yp++;
      errorTerm += k2;
    }
    else
      errorTerm += k1;
    xp += step;
    writePixel(xp, yp, color);
  }
}
```

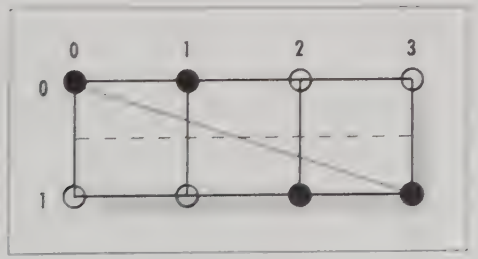


FIGURE 6-19
Locating the endpoint of the line

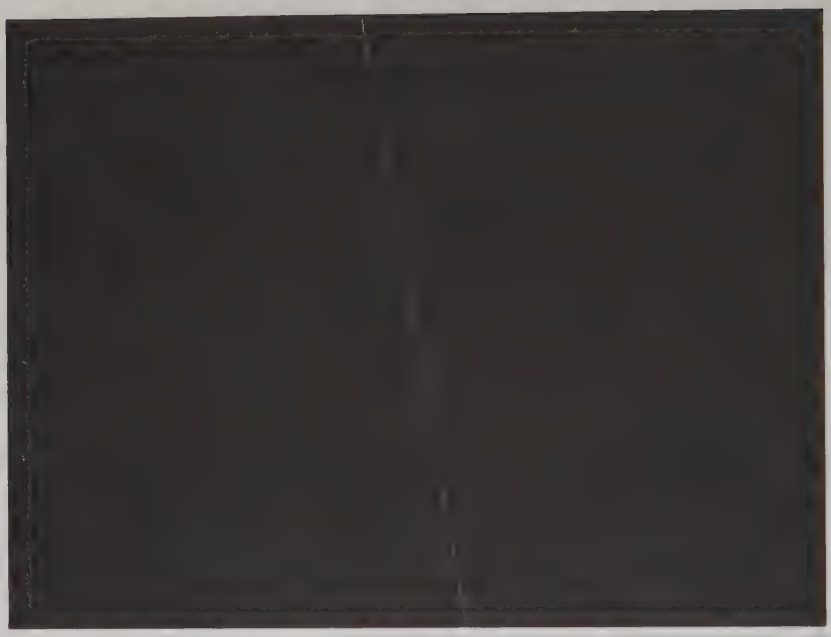


FIGURE 6-20
Screenshot of ApproxNextY() in action



Approximating the *x* Pixel

The complimentary function to ApproxNextY() is the function ApproxNextX(). This function must handle the case in which the absolute value of *dx* is less than (or equal to) *dy*. In this case, the *y* axis will be the primary axis and the *y* coordinate will be stepped successively as the algorithm approximates the *x* coordinate. An example of such a line orientation would occur if you wanted to plot the line with endpoints, P_1 (0,0) to P_2 (1,3), as shown in Figure 6-21.

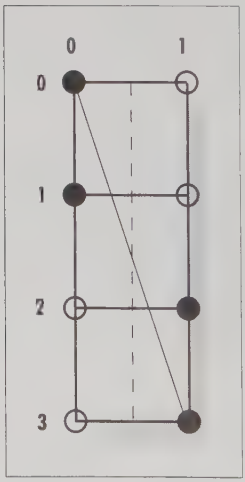


FIGURE 6-21
Orientation of a line where absolute |dx| >= *dy*

Such a line has the following initial conditions,

$dy = y_2 - y_1 = 3 - 0 = 3$
$dx = x_2 - x_1 = 1 - 0 = 1$

Just as three constants for the function ApproxNextY() were calculated, the same thing must be done here. Since the approximation is for the *x* axis, calculate the constants *k1* and *k2* by examining the variable *dx*. This looks exactly like computations for the approximation of the *y* axis only the variables *dx* and *dy* have been swapped.

```
k1 = dx * 2
k2 = 2 * (dx - dy)
errorTerm = k1 - dy
```

The line has, $P_1(0,0)$ to $P_2(1,3)$,

$k_1 = dx * 2 = 2 * 1 = 2$
$k_2 = 2 * (dx - dy) = 2 - (2*3) = -4$
$errorTerm = k_1 - dy = 2 - 3 = -1$

If the *errorTerm* is greater than or equal to zero, update the *x* coordinate, see Listing 6-11.

LISTING 6-11 Updating the *x* coordinate

```
if (errorTerm >= 0)
{
  errorTerm += k2;
  x++;
}
y+= step;
writePixel(x,y);
```

Otherwise, the *y* coordinate is incremented in the desired direction and the current *x* coordinate is used to plot the pixel. Listing 6-12 demonstrates how we update the *errorTerm* and the *y* coordinate.

LISTING 6-12 Updating the *errorTerm* and *y* coordinate

```
if (errorTerm < 0)
  errorTerm += k1;
y += step;
writePixel(x,y);
```



Looping Through the *y* Coordinates

By default, the algorithm immediately plots the coordinates passed into the function, see Figure 6-22(a). These are the endpoints to start the line so no approximation is necessary. The function is going to draw a line of length *dy* means it needs to approximate the *x* coordinates *dy* times. Simulating the first iteration of the loop with the example values, results in,

```
loop = 0  dy = 3  x = 0  y = 0  errorTerm = -1

errorTerm < 0
  errorTerm = errorTerm + k1 = -1 + 2 = 1                    errorTerm = 1
y = y + 1
y = 1
plot(x,y)
x = 0 y =1
decrement dx
dx = 2
```

The result of the first iteration of the loop can be seen in Figure 6-22(b). The first iteration of the loop approximates the best x coordinate as $x = 0$. The *errorTerm* was less than zero so the algorithm chose not to advance the x value. Upon completion of this decision the *errorTerm* was adjusted to a value greater than zero. For the second iteration,

```
loop = 1  dy = 2  x = 0  y = 1  errorTerm = 1

errorTerm > 0
  errorTerm = errorTerm + k2 = 1 + (-4) = 1                  errorTerm = -3
  x = x + 1
x = 1
plot(x,y)
x = 1 y =2
decrement dx
dx = 1
```

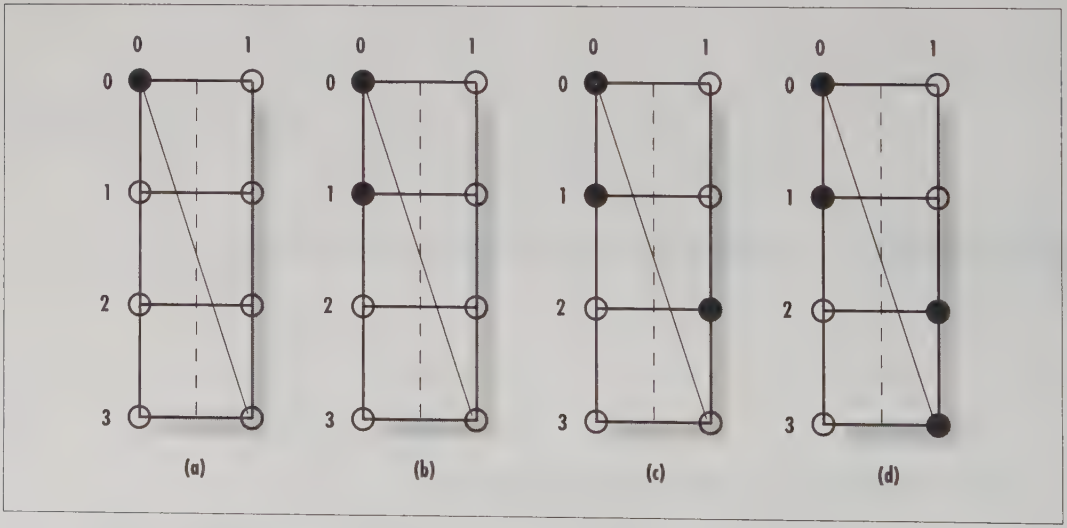


FIGURE 6-22 Stepping through each iteration of ApproxNextX()

During this iteration the *x* coordinate is advanced, thereby selecting the proper coordinate for the line. The result of this operation can be seen in Figure 6-22(c). The final endpoint is now calculated before the loop terminates.

```
loop = 2  dy = 1  x = 1  y = 2  errorTerm = -3

errorTerm < 0
  errorTerm = errorTerm + k1 = -3 + 2 = -1                    errorTerm = -1
y = y + 1
y = 3
plot(x,y)
x = 1  y =3
decrement dx
dx = 0
```

This completes the drawing of the line P1(0,0) to P2(3,1), as shown in Figure 6-22(d). The completed product can be seen in Figure 6-23. The source code for the Bresenham Line-Drawing algorithm used in *Spells of Fury* programs is presented in Listing 6-13.

LISTING 6-13 Source listing of the Bresenham Line-Drawing algorithm

```
void BresenhamLine(UINT x1, UINT y1, UINT x2, UINT y2, BYTE color)
{
   int dx, dy;
   int step;       // direction to proceed
   // if y1 > y2 swap coordinates
   if (y1 > y2)
   {
     swapTermsInt(&y1, &y2);
     swapTermsInt(&x1, &x2);
   }
   // calculate deltas
   dx = x2 - x1;
   dy = y2 - y1;
   if (dx > 0)
   {
     step = 1; // proceed in the positive direction
   }
   else
   {
     dx = -dx;
        step = -1;
   }
   if (dx > dy)
        ApproxNextY( x1, y1, dx, dy, step, color );
   else
        ApproxNextX( x1, y1, dx, dy, step, color );

}
```

continued from previous page

```
//////////////////////////////////////
// Approximate Next Y Pixel
//////////////////////////////////////
void ApproxNextY( UINT xp, UINT yp, UINT dx, UINT dy, int dirVector,
➥BYTE color )
{
  int k1, k2, errorTerm;
  k1 = dy << 2;
  k2 = k1 - (int)(dx << 2);
  errorTerm = k1 - (int) dx;
  writePixel(xp, yp, color);
  while (dx--)
  {
     if (errorTerm >= 0)
        {
          yp++;
          errorTerm += k2;
        }
        else
          errorTerm += k1;
        xp += dirVector;
        writePixel(xp, yp, color);
  }
}
//////////////////////////////////////
// Approximate Next X Pixel
//////////////////////////////////////
void ApproxNextX( UINT xp, UINT yp, UINT dx, UINT dy, int dirVector,
➥BYTE color )
{
  int k1, k2, errorTerm;

  k1 = dx << 2;
  k2 = k1 - (int)(dy << 2);
  errorTerm = k1 - (int) dy;

  writePixel(xp, yp, color);

  while (dy--)
  {
     if (errorTerm >= 0)
        {
          xp += dirVector;
          errorTerm += k2;
        }
        else
          errorTerm += k1;
        yp++;
        writePixel(xp, yp, color);
```

```
  }
}
////////////////////////////////////
// swap integers
////////////////////////////////////
void swapTermsInt( UINT *a, UINT *b)
{
  UINT temp;

  temp = *a;
  *a = *b;
  *b = temp;
}
```



Review of the Bresenham Line-Drawing Algorithm

All eight conditions of line orientations possible have now been satisfied for the Bresenham Line-Drawing Algorithm. Lines can easily be reclassified as having two possible orientations; a line whose slope is between zero and one, that is the absolute value of *dx* is greater than *dy* or a line that may have a slope greater than one, where the absolute value is less than or equal to *dy*. If the absolute value of *dx* is greater than *dy*, proceed along the *x* axis and approximate the best *y* coordinate. To handle lines where the absolute value of *dx* is less than or equal to *dy*, interchange the roles of the *x* and *y* coordinates



FIGURE 6-23

Screenshot of ApproxNextX() in action

Line-Clipping Concepts

If you're reading this book, then it's safe to assume that you have played at least one version of id Software's game *DOOM*. I believe I'm getting an affirmative nod of heads from my audience. Good. The *DOOM* games have a map you can use to examine the level you are playing in. It's a crude map made up of line segments, but nevertheless, serves its purpose to aid the players in getting an idea of their location and what's ahead; one such map is shown in Figure 6-24.

When you toggle between the texture mapped viewport and the 2D line-drawn map, you get a feel for the game's actual world coordinates. The map or world coordinates depict an area you can travel that is larger than the display is capable of showing. You can scroll left, up, down, or to the right, but you can't get the entire view of the map in one screen. What is happening is that the game is being described in a coordinate system more suited to processing the events for the game. The data type of the world coordinate system is most likely even different. For instance, the world coordinates are of the data type double or long, to handle the complex math required to do the texture mapping. The viewport coordinates are described as unsigned integers. The viewport is bounded, for example, by the dimensions of 640 pixels by 480 pixels, where the world coordinates for the game can be a Cartesian grid, of type double, for example, from -16000.00 to 16000.00. The map in the game *DOOM* only provides a snapshot of a region of the world coordinates. The code to display the world coordinates must understand how to translate the world coordinates to the viewport. You couldn't simply draw to pixel (-1000, 200); it must be mapped over to the viewport. In order to map the region for the snapshot, get

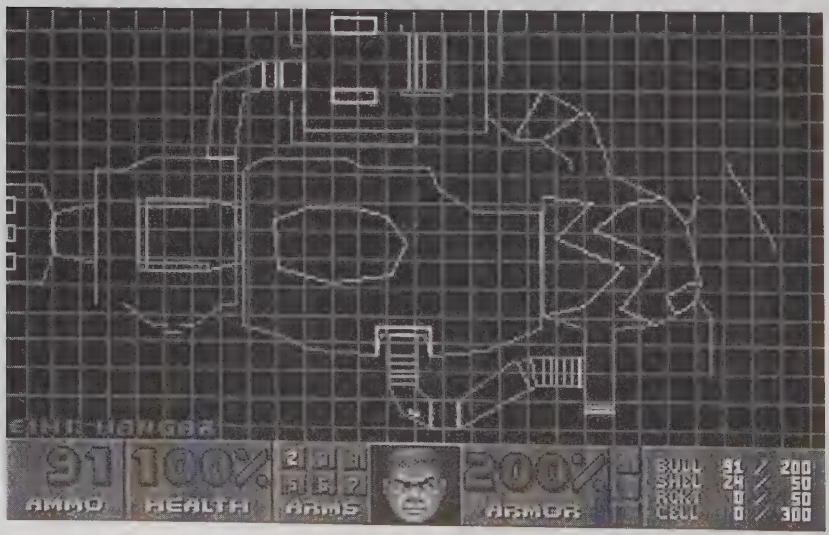


FIGURE 6-24
Polylines in a *DOOM* map

out the virtual scissors. But don't use your mom's good scissors! You must detect when a line, such as in the map for *DOOM*, leaves the snapshot and clip it off.



Line-Clipping Algorithms

Cutting lines that extend beyond the bounds of a window is called line-clipping. What usually occurs in a clipping algorithm is that the world coordinates are mapped over to the viewport where the lines are clipped against the viewport boundaries. It's useful to declare a boundary window to clip against. This will give more flexibility to the software and allow for variable sized windows to clip against in the viewport. Now define a boundary using the following unsigned integers,

```
UINT wx_max, wx_min, wy_max, wy_min;
```

These four values will allow testing of the endpoints of a line to determine if they are within the boundaries of a window, or if they lie outside. An endpoint that lies within the boundaries satisfies the inequality,

$wx_{min} < x < wx_{max}$ and $wy_{min} < y < wy_{max}$

If the value of x is to the left of wx_{min} or to the right of wx_{max} the line is clipped. The same condition exists for the y coordinate. If y is above wy_{min} or below wy_{max}, the line is also clipped.

Figure 6-25 shows four examples of lines before and after they are clipped against the window boundary. The line with endpoints P_1P_2 is discarded because its endpoints lie completely outside the window. The line with endpoints P_3P_4 is drawn, unclipped, because its two endpoints are completely within the window boundary. The line P_5P_6 only clips endpoint P_6 that is to the

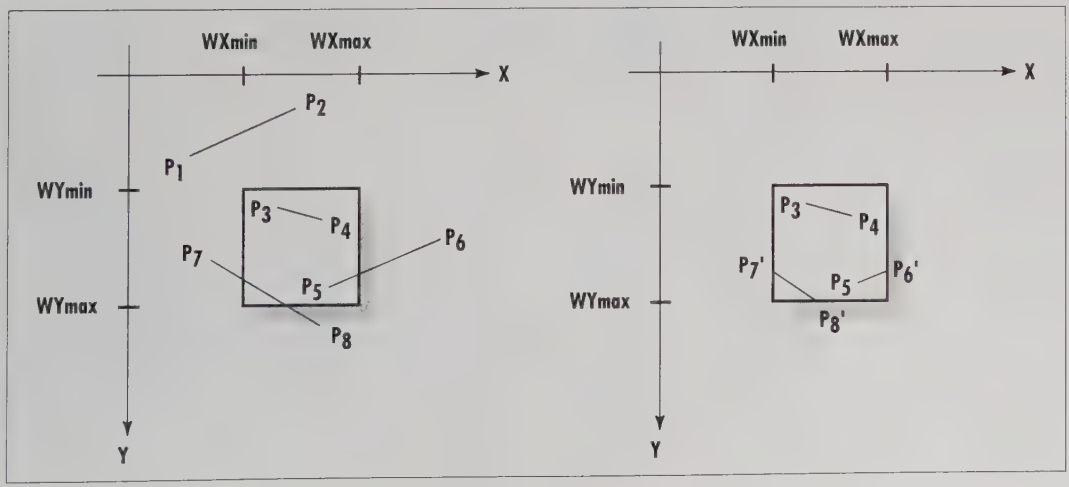


FIGURE 6-25 Clipping against a window boundary

right of the window boundary. The endpoint P_5 is inside the boundary, so it is left alone. The last line, described by endpoints, P_7P_8, has both of its endpoints clipped; they both are outside the window boundary. The endpoints P'_6, P'_7, and P'_8 were all clipped at the intersection of the window boundary.

The line-clipping algorithm must examine each line's endpoints and determine which points it will accept, clip, or reject. Expensive SGI workstations have the hardware to perform this operation. On the other hand, using PCs means picking an algorithm that is suitable for the limitations of the computer being used. The PC market is almost to the point where it can assume that nearly everybody does or will have a floating-point processor. The sophistication of games today with complex polygon generation almost demands this little piece of hardware. Most applications that will perform line-clipping may do so on the order of magnitude of thousands of line segments per screen update. To minimize the number of line segment computations, throw out the lines that won't be displayed in the window. Also, recognize that line segments that do not need clipping are located within the window. This minimizes the case for computation so only lines that need to be fully or partially clipped need to be handled.



The Cohen-Sutherland Line-Clipping Algorithm

The Cohen-Sutherland line-clipping algorithm uses a coding scheme where endpoints are analyzed and tagged with a region code. The region code is used to determine if the window boundary intersection computations can be avoided. The regions are set up to identify window boundaries, as shown in Figure 6-26.

Since this is the age of recycling, use the old endpoints from Figure 6-25, and encode their regions. Figure 6-27 shows the results of encoding these line segments. The line P_1P_2 is completely above the window boundary. Point P_1 is above and to the left of the window, and P_2 is above it. The line segment P_1P_2 is rejected for any further processing. The line segment P_3P_4 is completely inside the window boundary. It is trivially accepted and drawn to the window with no further computation. The line segment P_5P_6 is a case where P_5 is completely inside the window and P_6 is to the right. The point P_6 is identified as a point to the outside and locating the boundary intersection is required. The last line segment, P_7P_8 is the worst-case line. This line passes completely through the window and its endpoints lay outside the window boundary. The point P_7 is encoded as left of the window boundary, and P_8 as below. Note the distinction between this line segment and P_1P_2, which was rejected. Computations for the window boundary intersection for both endpoints of the line segment P_7P_8 are required.

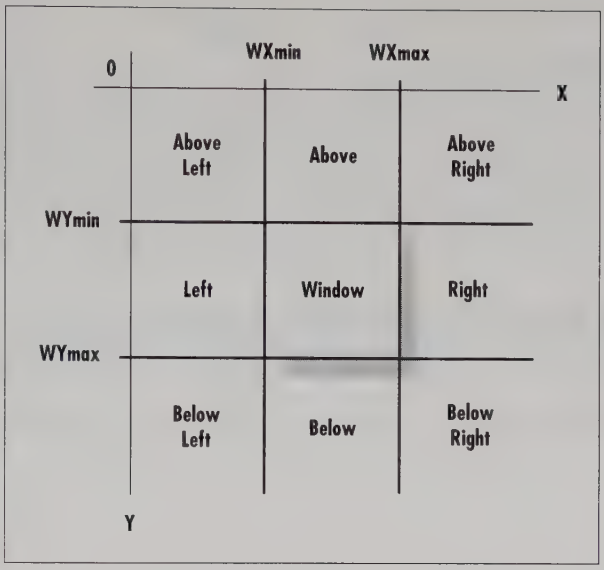


FIGURE 6-26
Cohen-Sutherland region codes

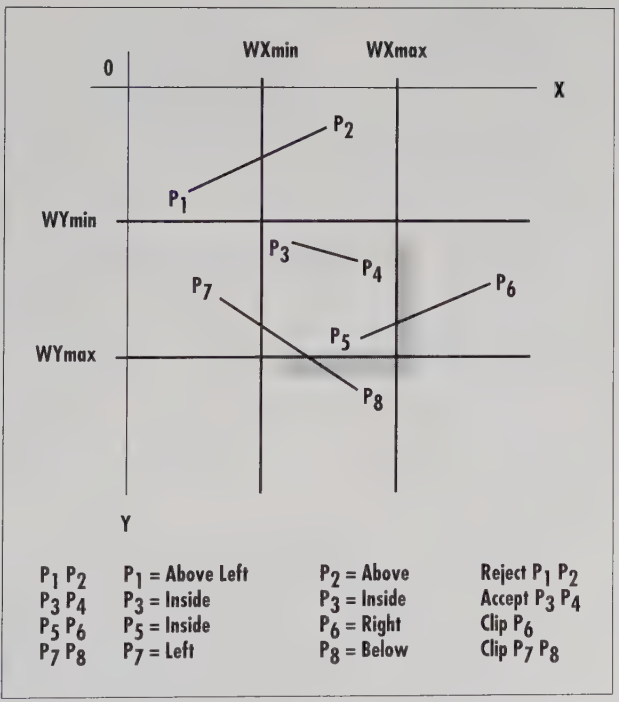


FIGURE 6-27
Region coding endpoints



Encoding the Endpoints

The Cohen-Sutherland algorithm minimizes computation by dividing the window boundary clip region into nine boundaries. To code this algorithm provide a means to define a window boundary. This is a trivial function but nevertheless an important one. The function SetClipBounds(), shown in Listing 6-14, is used to set the window boundary.

LISTING 6-14 The SetClipBounds() function

```
UINT wx_max, wx_min, wy_max, wy_min; // Cohen-Sutherland coordinates

void SetClipBounds( UINT left, UINT top, UINT right, UINT bottom)
{
  wx_min = left;
  wy_min = top;
  wx_max = right;
  wy_max = bottom;
  wRect.left = left; wRect.top = top;
  wRect.right = right; wRect.bottom = bottom;
}
```

The window boundaries are declared as the global variables *wx_min*, *wy_min*, *wx_max*, and *wy_max*. This simple function provides all the information that is needed to commence encoding the line segments. Depending on how old your college text is, you may have a listing which uses a binary array or an array of Boolean codes for a Pascal implementation. In C/C++ just use an array of Boolean entries.

```
#define MAX_ENCODE_ENTRIES 4     // Cohen-Sutherland Encoding bits
BOOL codeP1[MAX_ENCODE_ENTRIES], codeP2[MAX_ENCODE_ENTRIES];
```

Anybody care to take a guess why one point has a Boolean array of four values? Well, first off, each index into the array is of an enum value,

```
enum encodeBits {
   LEFT,       // x < xmin (left)
   RIGHT,      // x > xmax (right)
   BOTTOM,     // y > ymax (below)
  ABOVE        // y < ymin (above)
};
```

The Boolean array encompasses all the possible values for the nine region codes as demonstrated in Figure 6-28. An endpoint with region code BELOW-RIGHT will have its Boolean code array values, codeP1[RIGHT] = TRUE and codeP1[BELOW] = TRUE. To test possibly thousands of points it is first necessary to initialize the Boolean encode array to all FALSE. The code for initCodes() is shown in Listing 6-15.

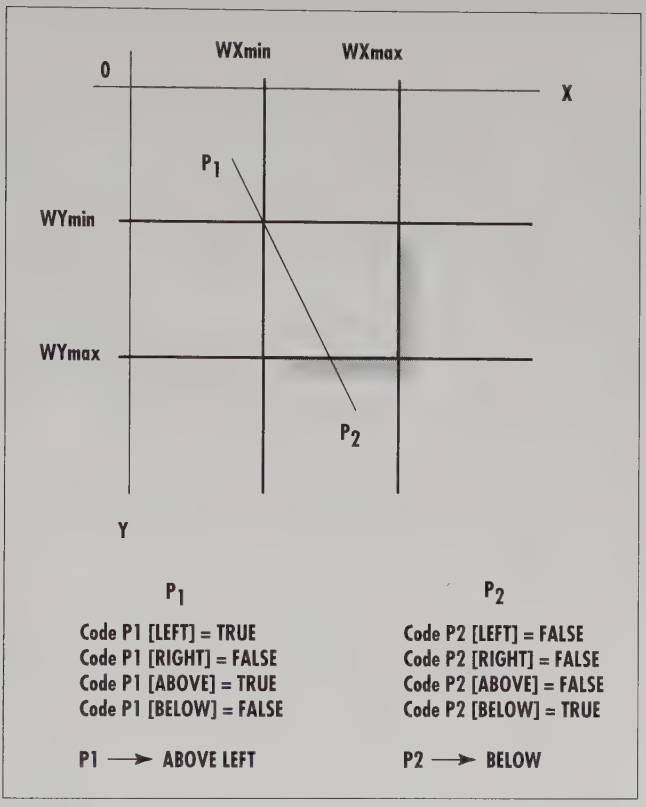


FIGURE 6-28

Using a Boolean array to encode points P_1P_2

LISTING 6-15 Initializing the encoding logic arrays

```
void initCodes()
{
  UINT index;

  for (index = 0; index < MAX_ENCODE_ENTRIES; index++)
  {
    codeP1[index] = codeP2[index] = FALSE;
  }
}
```

The code for encoding the endpoints is almost as simple. Listing 6-16 shows the code for function encode().

LISTING 6-16 Encoding the logic arrays

```
void encode( UINT x, UINT y, BOOL *code )
{
  // is the end point left of the window?
  if (x < wx_min)
```

continued on next page

continued from previous page

```
      code[LEFT] = TRUE;
   else
      code[LEFT] = FALSE;

   // is the end point right of the window?
   if (x > wx_max)
      code[RIGHT] = TRUE;
   else
      code[RIGHT] = FALSE;

   // is the end point above the window?
   if (y < wy_min)
      code[ABOVE] = TRUE;
   else
      code[ABOVE] = FALSE;

   // is the end point below the window?
   if (y > wy_max)
      code[BOTTOM] = TRUE;
  else
      code[BOTTOM] = FALSE;
}
```

An endpoint's *x* and *y* coordinates are passed into the function as unsigned integers along with a pointer to the Boolean array that will contain the region codes. The Boolean array of endpoint codes will tell if the line segment lies inside the window or outside the window. A Boolean array with codes all set to FALSE indicates an endpoint that lies completely inside the window boundary. If the second endpoint is encoded using Boolean array, *codeP2* is also set entirely to FALSE, then the entire line segment is inside the window.



Dealing with Rejection

Test to see if a line segment is completely inside the window by using the function trivialAccept(). This is accomplished by looping through both Boolean code arrays for the two endpoints. If none of the codes in either code array are set to TRUE, then the line is trivially accepted (see Figure 6-29). This means the line segment can be drawn to the window as is with no line intersection computations. The code for function trivialAccept() is shown in Listing 6-17.

LISTING 6-17 Trivially accepting a point code

```
BOOL trivialAccept(BOOL *code1, BOOL *code2)
{
  UINT index;

  for (index = 0; index < MAX_ENCODE_ENTRIES; index++)
  if ( (code1[index] == TRUE) || (code2[index] == TRUE) )
```

```
    return FALSE;

  return TRUE;
```

In a similar fashion, examine the line segments and determine if they can even be drawn in the window at all. Use the function reject() to determine this. This test is conducted by stepping through the code array of both endpoints simultaneously. If a region code is set to TRUE at the same location in any of the two arrays, reject the line segment, as shown in Figure 6-30. The code for function reject() is shown in Listing 6-18.

LISTING 6-18 Rejecting a point code

```
BOOL reject(BOOL *code1, BOOL *code2)
{
  UINT index;

  for (index = 0; index < MAX_ENCODE_ENTRIES; index++)
    if ( (code1[index] == TRUE) && (code2[index] == TRUE) )
      return TRUE;

  return FALSE;
}
```

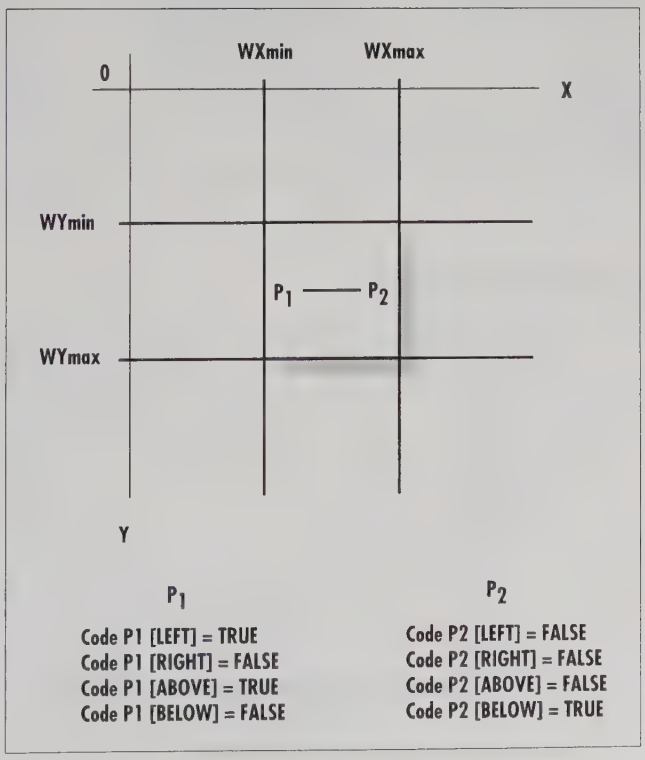


FIGURE 6-29
A case for trivially accepting a line segment

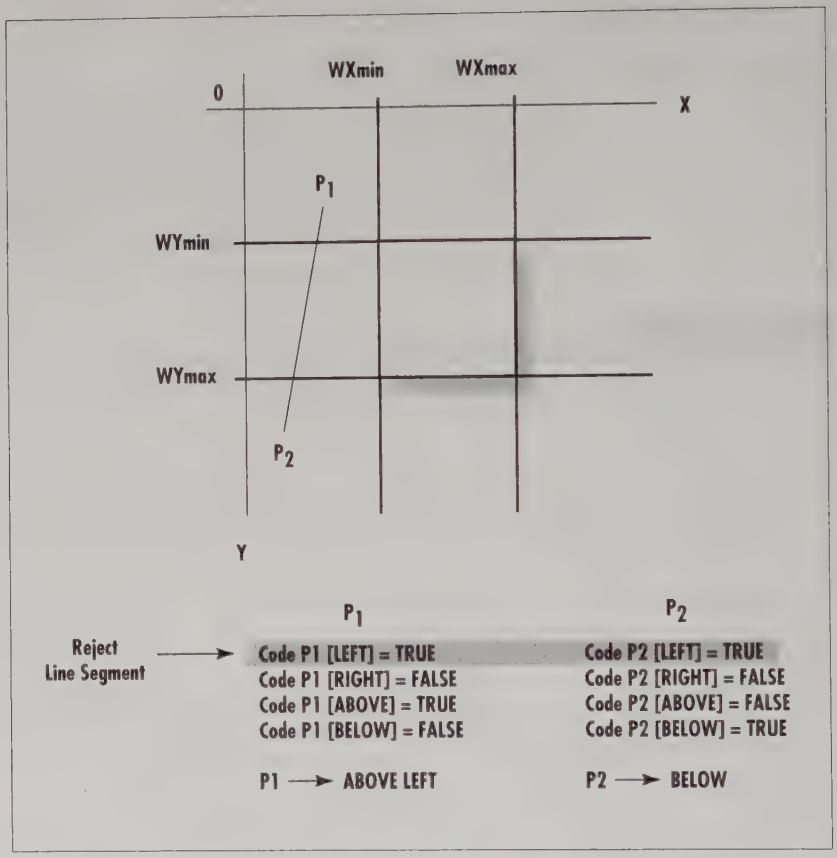


FIGURE 6-30
Case for a trivial rejection

Checking for Intersections

The example shown in Figure 6-31 shows the line segment P_1P_2 that is not completely inside the window or completely outside of it. The figure also shows the corresponding region codes for the endpoints P_1 and P_2. At this point, the line has been tested for conditions at the window boundaries, and the line segment can be neither trivially accepted nor rejected. Now determine where an endpoint is from a boundary and how to clip the remainder of the line against the window boundary.

The algorithm works in an ordered manner. It first processes the endpoints that are to the left, then the right, those below, and finally above. From the Boolean code arrays see that the endpoint P_2 is below the window boundary and the endpoint P_1 is above it. Following the algorithm's ordered manner, start with P_2. The algorithm will determine the intersection of the line segment P_1P_2 against the lower window boundary. The point on the boundary will be P'_2. The line segment from P'_2P_2 will be discarded, as shown in Figure 6-32.

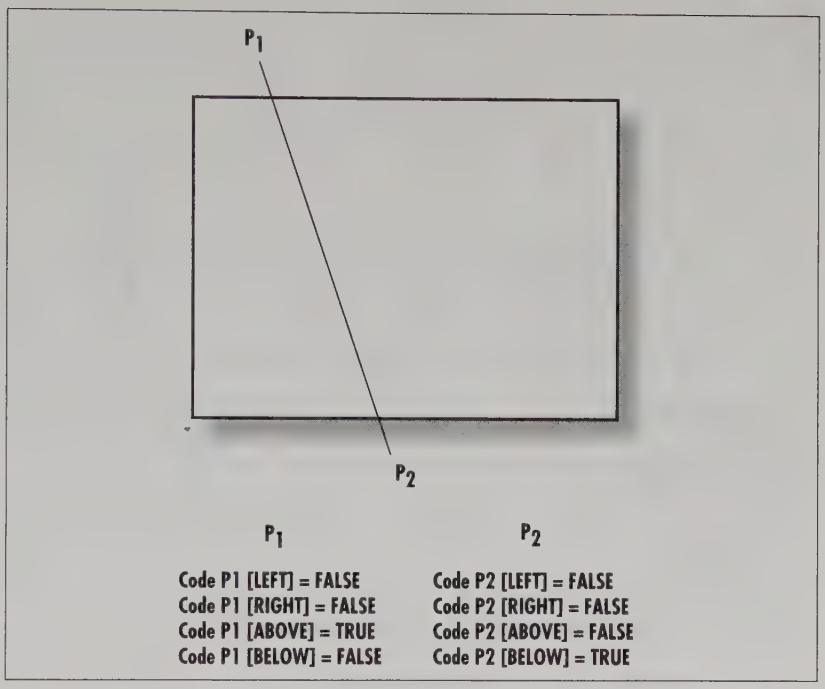


FIGURE 6-31

A line crosses the window boundaries

The line segment P_1P_2 has now been reduced to $P_1P'_2$. The region codes for P_1 are now checked, and the algorithm determines P_1 is above the window. The intersection, point P'_1, is now calculated and the line segment P_1P_1' is discarded. The line has now been properly clipped to the window boundary, see Figure 6-33.

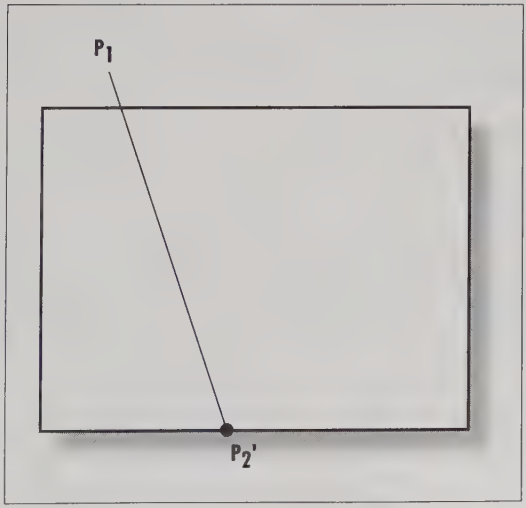


FIGURE 6-32

Clipping a line segment to the lower-window boundary

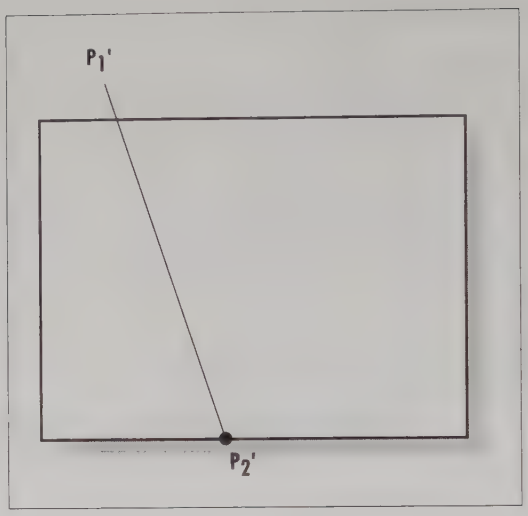


FIGURE 6-33
Clipped line segment $P'_1P'_2$

Calculating Points of Intersection

Earlier in this chapter you saw the slope intercept equation, $y = mx + b$. Revisit this equation and use it to determine the intersection points of our line segments with the window boundary. You already know that a line segment is described by two points, P_1 and P_2. P_1 and P_2 have (x,y) coordinates, $P_1(x_1,y_1)$ and $P_2(x_2,y_2)$. We can compute the y intercept with a vertical boundary using,

$$y = y_1 + m(x - x_1)$$

where

$$x = wx_{max} \text{ or } wx_{min}$$

Similarly, we can calculate the x intercept with a horizontal window boundary by using,

$$x = x_1 + (y - y_1)/m$$

where

$$y = wymax \text{ or } wymin$$

Now you're probably asking yourself, how do you know which equation to use and when? Well, our friends the region codes lend a hand here. Following the algorithm executes in a fixed order, examine the array value of code[LEFT] first; if this cell of the array is TRUE, then,

$$y = y_1 + m(wx_{max} - x_1)$$

If the value in this cell is FALSE, proceed to check the Boolean value in code[LEFT]. If code[LEFT] is TRUE,

$$y = y_1 + m(wx_{min} - x_1)$$

In the event code[LEFT] contains FALSE then check code[BOTTOM]. If this is set to TRUE, then use,

$$x = x_1 + (wy_{max} - y_1)/m$$

And finally, test code[ABOVE]. If TRUE, then use,

$$x = x_1 + (wy_{min} - y_1)/m$$



The pre-ordered mannerism of the Cohen-Sutherland line-clipping algorithm doesn't make it the most efficient clipping algorithm. However, its uses are still adequate on the PC where, for now, not all hardware is capable of performing clipping in the video card.

Well, enough of the math for right now. Your head's probably full. You want to know how to implement this beast right? Well, start off by encoding both endpoints of the line segment and see if you can trivially accept or reject the line. If you can't perform either of these it's time to start calculating intersections. A trick used in this code is to always insure that the endpoint P_1 is outside the window segment. That is if a line were clipped where P_1 is inside the window and P_2 is outside, you can swap endpoints and recode the region's arrays. From then on look at the region codes and determine which equation to use on the endpoint to find the intersection with the window boundary (see Figure 6-34). The code for the Cohen-Sutherland line-clipping algorithm is presented in Listing 6-19.

LISTING 6-19 The Cohen-Sutherland line-clipping algorithm

```
void ClipLine( UINT x1, UINT y1, UINT x2, UINT y2, BOOL *code1, BOOL *code2)
{
  BOOL done = FALSE;
  BOOL display = FALSE;
  BOOL *code;
  double m;
  double fx1, fy1, fx2, fy2;

  code = code1;
  while(!done)
  {
    initCodes();
    encode(x1, y1, code1);
    encode(x2, y2, code2);
    if (accept(code1, code2) == TRUE)
    {
      done = TRUE;
      display = TRUE;
      break;
```

continued on next page

continued from previous page

```
    }
    else if (reject(code1, code2) == TRUE)
    {
      done = TRUE;
    }
    else  // find intersection
    {
      // do we need to swap the points?
      if ( (code1[LEFT] == FALSE) && (code1[RIGHT] == FALSE)
          && (code1[ABOVE] == FALSE) && (code1[BOTTOM] == FALSE) )
      {
        // yes? swap P1 and P2
        swapTermsInt(&y1, &y2);
        swapTermsInt(&x1, &x2);
        initCodes();
        encode(x1, y1, code1);
        encode(x2, y2, code2);
      }
      // calculate the slope (m)
      fx1 = (double)x1; fy1 = (double)y1;
      fx2 = (double)x2; fy2 = (double)y2;
      m = (fy2-fy1) / (fx2-fx1);

      if (code1[LEFT] == TRUE)  // point is to the left
      {
        y1 = (int) (fy1 +((double)wx_min -fx1) * m);
        x1 = wx_min;
      }
      if (code1[RIGHT] == TRUE) // point is to the right
      {
        y1 = (int) (fy1 +((double)wx_max -fx1) * m);
        x1 = wx_max;
      }
      if (code1[BOTTOM] == TRUE)  // point is below
      {
        x1 = (int) (fx1 + ((double)wy_max - fy1) / m);
        y1 = wy_max;
      }
      if (code1[ABOVE] == TRUE) // point is above
      {
        x1 = (int) (fx1 + ((double)wy_min - fy1) / m);
        y1 = wy_min;
      }
    } // find intersection
  } // while NOT done
  if (display == TRUE)
  {
    BresenhamLine(x1,y1,x2,y2, 164);
  }
}
```

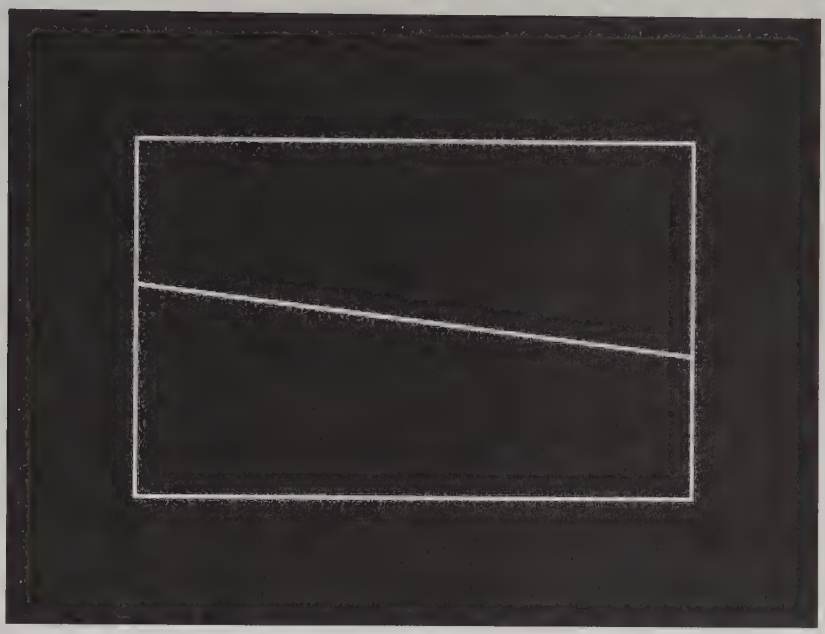


FIGURE 6-34
Screenshot of Cohen-Sutherland line-clipping using DirectDraw

Cohen-Sutherland Line-Clipping Algorithm in Review

The Cohen-Sutherland algorithm tests lines to determine if a calculation can be avoided. The line's endpoints are assigned region codes that are examined to see if the line can be trivially accepted or rejected. If a line cannot be determined to be completely inside a window or outside of one, the line must be checked to see where it encounters a boundary. The line segment from the boundary intersect to the endpoint is discarded. These intersections are computed using the slope line equations. The algorithm checks line endpoints against window boundaries in the order of left, right, top, and bottom. Because of this ordered testing, the Cohen-Sutherland line-clipping algorithm is only moderately efficient.

Polygons

The Euclidean definition of a polygon is a closed shape composed of three or more edges. Edges of polygons are line segments which connect to form a two-dimensional closed shape. Triangles, pentagons, hexagons, and octagons are examples of polygons. Polygons can be classified as being convex or nonconvex. A polygon is convex if a line is drawn between two interior points, and this line encounters only the interior of the left edge and/or the interior of the right edge.

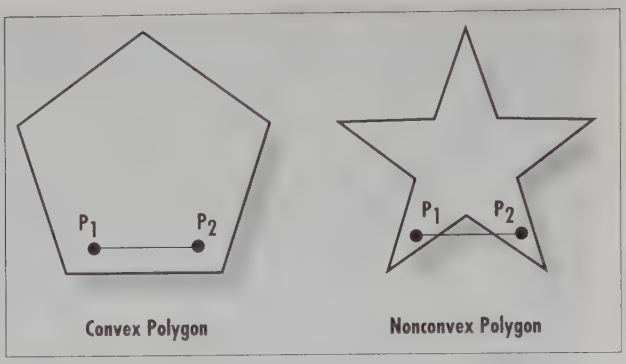


FIGURE 6-35
Convex and nonconvex polygons

If a line is drawn from two interior points, and the interior of an edge is encountered, as well as the exterior, the polygon is nonconvex. This distinction is made in Figure 6-35.



Describing a Polygon

Probably the most fundamental polygon ever blitted to a game screen is the spaceship from the old Atari *Asteroids* coin-op game. The ship was simple in shape, almost a triangle, but the ship had fins. Like most arcade games, the ship moved around in a world coordinate system and was translated over to the game screen. The player's spaceship itself was logically mapped to a Cartesian coordinate grid around the origin, as shown in Figure 6-36. It followed a dot-to-dot scheme, in the left direction from the first vertex, at point (-8,0), giving the polygon shape a negative orientation. Connecting the vertices forms a network of closed edges and a simple polygon spaceship takes form.

Describing the spaceship takes a collection of vertices. Each vertex is a set of coordinates. At this point in the game, the ships needs to be rotated and moved. This means a good amount of computation is required, which requires a system of points suitable to describe the ship's flight path. This is jumping ahead a bit, but I just want to emphasize the reason for defining our own point data structure rather than using the one provided by the Windows API. This definition will need a coordinate system that is suitable for adjusting to changes in rotation, and this is shown in Listing 6-20.

LISTING 6-20 Definition of the point structure

```
typedef struct
{
  double x;    // x coordinate
  double y;    // y coordinate
} Point;
```

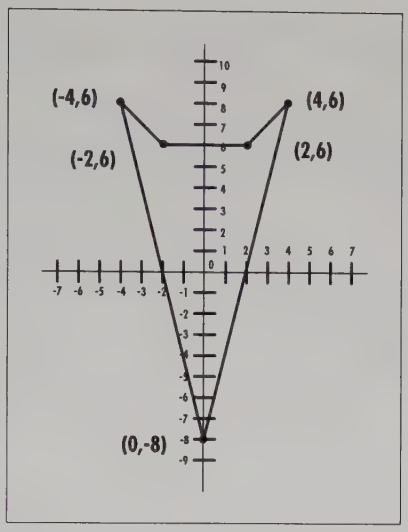


FIGURE 6-36
Vertices of a simple spaceship

This simple data structure will allow a definition of the *Asteroids* spaceship's shape. Using the polygon shape depicted in Figure 6.36, we will play dot-to-dot and map our shape. This means we will first start with the vertex on the x=0 coordinate and proceed in a clockwise manner gathering the vertices to form an array of Points. The figure has

```
Point spaceshipArray[] =
{
  {0,-8},{-4,8},{-2,6},{2,6},{4,8}
};
```

The assumption is that the last coordinate in the array is the endpoint for the last edge closing the polygon. This means the last set of vertices are actually $v_1(4,8)$ to $v_2(0,-8)$. To retrieve these coordinates use the code in Listing 6-21.

LISTING 6-21 Defining a polygon spaceship

```
Point v1, v2;

v1.x = spaceshipArray[4].x;
v1.y = spaceshipArray[4].y;
v2.x = spaceshipArray[0].x;
v2.y = spaceshipArray[0].y;
```

Now there is a means to statically define a shape. As it stands, this isn't very interesting nor would the computer superstore purchaser give the time of day to present this product. The shape needs to be interesting and give it smooth movement. For the eager novice who has some experiencing blitting, this discussion may seem a bit lame, or out of date. After all, *Mortal Kombat* doesn't use lines or polygons. The usage of polygons in game design is in its own little

renaissance. In the 1980s, blitting sprites was the de facto standard for game design and implementation. In the late 1980s and early 1990s, the UNIX super workstations started making their debut. The game developer was soon introduced to a tool far beyond the capabilities of even the most powerful of PCs. Old 3D games using lines and polygons, like *Battle Zone* and *Star Wars* for the UNIX workstations, were retrieved from their archives and given texture mappings. Now with the advent of the Pentium processor, the PC is almost touching the tip of the iceberg in computational performance. What is seen in the industry now is a movement away from digitized warriors and a move toward polygon-mapped virtual warriors from high-powered workstations. Programmers are finding themselves thrown back into the complex programming arena of working with line segments and polygons, and sorry to say, complex geometric mathematics. The large game companies are already harnessing the computing power of SGIs for generating their animation. As Homer Simpson so richly put it, "I'm not giving in to conformity, I'm just hopping on the band wagon." So what are you waiting for? Take a deep breath, walk over to your manager's office and demand that the company sign the $150K purchase requisition for the workstation.



Basic Transformations and Polygon Motion

Animating polygon objects on the game screen requires producing the illusion of smooth motion. The objects to animate are defined by a set of coordinates. As the objects are moved, new coordinates are calculated and these coordinates are then displayed on the screen. The process of moving object coordinates around on a game screen is called geometric transformation. The fundamental transformations are translation and rotation. When an object is moved on the screen, geometric transformation functions are called to move the polygon to its new set of coordinates.



Translation of a Polygon

The simplest arithmetic operation that can be performed on a set of vertices is a distance translation. To move the object to the right by six pixels use the code in Listing 6-22.

LISTING 6-22 Moving vertices six pixels to the right

```
int index;

for (index = 0; index < MAX_VERTICES - 1; index++)
```

```
{
  spaceshipArray[index].x + 6; // shift space ship to right 6 pixels
}
```

In the same fashion, to make the ship appear to move up the game screen subtract a number of pixels from the *y* values in the objects vertices list (an array). This would translate the spaceship in the upward direction, see Listing 6-23.

LISTING 6-23 Moving vertices 16 pixels up the *y* axis

```
int index;
for (index = 0; index < MAX_VERTICES - 1; index++)
{
  spaceshipArray[index].y - 16;  // move space ship upward 16 pixels
}
```

An important point to establish here is that the coordinates are logical. Look how the spaceship was moved upward by subtracting 16 pixel units. In the logical Cartesian grid, we just pushed the spaceship down the axis in the negative *y* direction by 16 units. Didn't mean to confuse you so early, but this is just to make you aware of some of the headaches you're going to have when you debug this stuff. I think as the game industry makes a shift to the 3D polygon engines, the stock in headache pharmaceuticals is going to rise proportionally.

The primary usage of translations is to map logical off screen objects, which are rotating, computing, and having other hideous things done to them. Once done, the object can be translated to the game screen as shown in Figure 6-37.

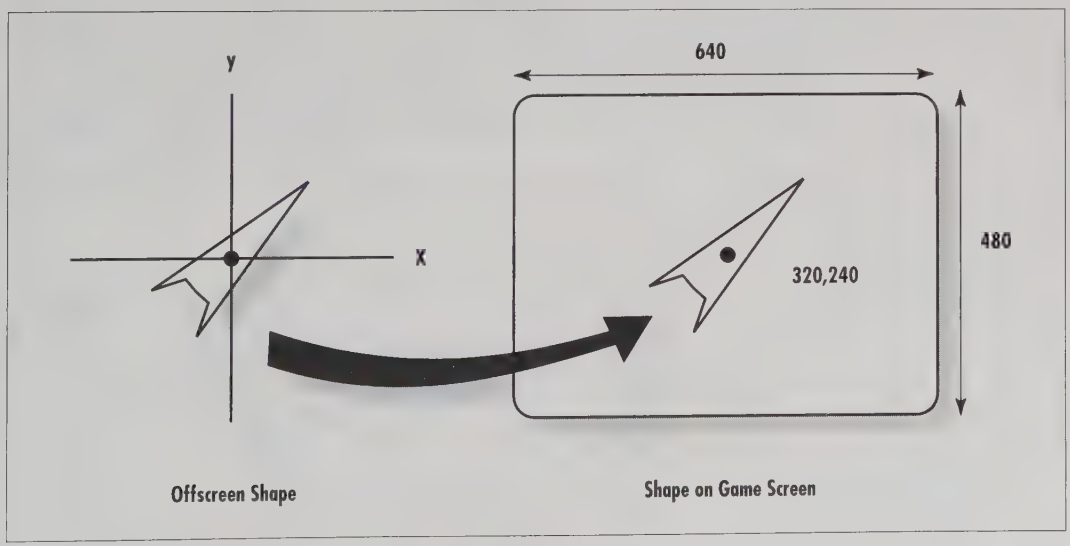


FIGURE 6-37 Translation of an offscreen object

Polygon Rotation

Changing the orientation of a polygon based on an angular direction is called rotation. The amount of rotation is specified by a rotation angle. Each vertex in the polygon is adjusted to rotation. The displacement of a vertex, moving point P(x,y) to a new point P'(x',y') is calculated about the origin of the logical coordinate system based on the angle of rotation, see Figure 6-38.

Angles on the computer are measured in radian units. One angular degree is equivalent to pi divided by 180 degrees. So one radian unit is equal to 0.01745 degrees. To rotate a vertex use the code shown in Listing 6-24.

LISTING 6-24 Rotating a vertex

```
#define RADIAN_UNIT 0.01745

double radianAngle;
double newVertexX, newVertexY;
int index;

radianAngle = rotationAngle * RADIAN_UNIT;

newVertexX = spaceship[index].x * cos(radianAngle)
                    - spaceship[index].y * sin(radianAngle);

newVertex = spaceship[index].y * cos(radianAngle)
                    + spaceship[index].x * sin(radianAngle);
```

Positive values for the *radianAngle* indicate a counter-clockwise rotation. Likewise, negative values indicate a clockwise rotation. The code performs a rotation about the origin of the logical grid. As you can see, the rotation of a polygon is mathematically intensive on the computer. It was for this reason that polygons were avoided on the PC for some time. Animation of objects

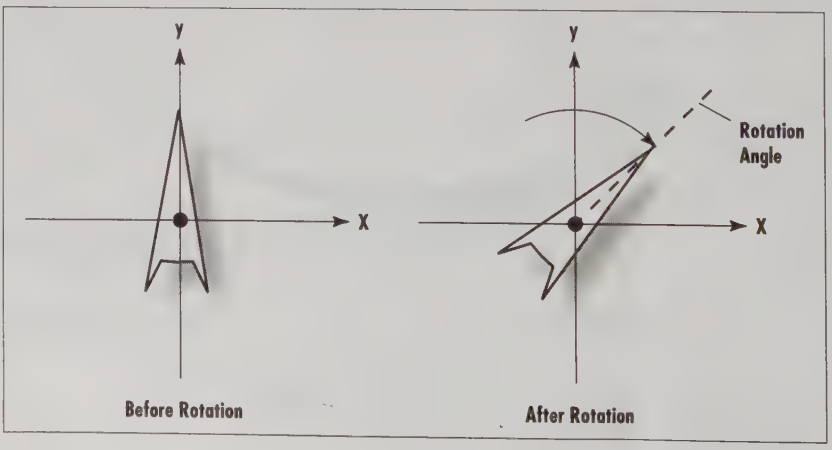


FIGURE 6-38
Rotating the polygon object around its origin

composed of thousands of polygons can become excessive and inhibit the performance of a PC. You can use that last line as a justification on your SGI purchase requisition. If it works, let me know: I haven't tried that one yet. I do know that none of the manager buzzwords work for purchases over $5K.



OOP—Object-Oriented Polygons

So far, the discussion has focused on polygons and geometric operations that can be performed on the vertices. The vertices of the polygon are nothing more than a list of points defining the shape of the polygon. Translations and rotations are the primary operations used on the list of vertices. These relationships of the polygon can be encapsulated to create a polygon class, called Shape. The class declaration is shown in Listing 6-25.

LISTING 6-25 Shape class definition

```
class Shape {
  protected:
    Point *ShapeDef;     // list of vertices to define polygon
    Point *sprite;       // vertices list describing a screen object
    int   pointEntries; // number of vertices in the list

    Point screen;        // screen coordinates
    Point vector;        // velocity vector
    int   angle;         // angle of rotation

  public:
    Shape();             // default constructor
    Shape& operator=( const Shape& s ); // overld assignment operator
    void encodeObject( Point *pointArray, int objectSize);
    void rotateShape( int shapeAngle );
    void compositeTransformation();
    void renderShape();
    ~Shape();            // default destructor
};
```

Actually, this declaration is the *Reader's Digest* version. The actual declaration is considerably longer although the key member functions and data members are listed above. Recall from the torturous discussion on DIBs that the reserved word protected is used for data hiding. That's being stingy with the code. It's a good development practice in a multi-developer environment to lock out as much code as possible from other developers (or from yourself). The benefits return ten-fold on this when a screw-up, uh-hum, a feature is discovered, and your group needs to release a fix. This minimizes the number of development groups outside of your own who are angry with you.



Building the Asteroid Ship with Our Shape Class

The Shape class makes it easy to build a polygon spaceship from the Point array. You need to declare the object, initialize it, and you're off and running. The code for this is shown in Listing 6-26.

LISTING 6-26 Defining a spaceship from the point array

```
Shape *spaceShipObject = new Shape;  // declare and initialize object

int size = sizeof(spaceshipArray[0]);
int objectSize = sizeof(spaceshipArray)/size;
spaceShipObject->encodeObject( spaceshipArray, objectSize );
```

Now the object consists of a list of vertices. The member function encodeObject() takes the array of points as input and uses these points for the object's data member *shapeDef*. The Shape class data member *shapeDef* is the polygon object's list of vertices. If you're confused by the two sets of vertices *shapeDef* and *sprite*, just imagine that the *shapeDef* is the master polygon image, and the *sprite* is scratch paper you're going to use for translations and rotations. For example, a rotation, using the member function rotateShape(), is shown in Listing 6-27.

LISTING 6-27 Shape member function rotateShape()

```
Shape::rotateShape( int shapeAngle )
{
  double radianAngle;
  int    index;

  radianAngle = shapeAngle * RADIAN_UNIT;
  for (index = 0; index < pointEntries; index++)
  {
    sprite[index].x = shapeDef[index].x * cos( radianAngle )
                    - shapeDef[index].y * sin( radianAngle );
    sprite[index].y = shapeDef[index].x * sin( radianAngle )
                    + shapeDef[index].y * cos( radianAngle );

}
```

The code is nearly identical to that presented in the previous section on rotating polygons. As you can see, the vertex list for the sprite object is dynamic. It changes with each rotation and transformation. The vertex list in *shapeDef* is static; that is, don't mess with it. Of course, the rotation is performed about the origin of the logical grid. The list of vertices in the data member sprite now needs to be translated to the game screen. The translation can be handled in the following manner,

```
int index;

for (index = 0; index < pointEntries; index++)
{
  sprite[index].x += screen.x;
  sprite[index].y += screen.y;
}
```



SpaceFoil: Using Polygon Objects

SpaceFoil is actually an old game called *SpaceWar* that's been around forever. Well, forever in PC years anyway. It was first introduced as shareware, or possibly it was even a coin-op. It was available on both the XT and for UNIX workstations available on the net. *SpaceWar* was a two-player game, where the players would try to destroy each other's ships. The ships were simple vector graphics quality. Vector graphics is a now defunct computer buzz word. It meant polygon line-drawn computer graphics. More interesting flavors of *SpaceWar* had stars or black holes in the center of the screen. The object of this particular game was to stay out of the gravity well and kill your opponent. In recent years, a more colorful release of *SpaceWar* I believe was named *Star Control*. This bitmapped version was very clever. One of my favorite ships to pilot in this game was manned by *sirens*. The sirens would call the crew from the opponents' ship and leave it stranded and helpless. This game is probably still available in one of those sordid, back-alley, discount software stores. My rendition is called *SpaceFoil*. I attempted to make a *Star Fury* ship from the television show *Babylon 5* and the *SpaceFoil* is what came out instead. *SpaceFoil* is vector graphics-based. I know, I know; some of you are back there snarling at the fact it's 2D and only composed of lines. In the UNIX workstation world, as I said earlier, these kinds of games are going through a rediscovery period. *Asteroids, Battle Zone,* and *SpaceWar* are coming back on the net for Suns, SGIs, and DECs. They are simple fun games that only require a tweak here or there to create a new game. The flip side to this revolution is to look at how much money Hollywood has pumped into entertainment CD-ROM, big budget software and have yet to crank out a classic fun-to-play game. For some reason these simple vector graphics games are challenging and addicting to their players. It's going to take a clever developer to come along, like the case with *Star Control*, and add a new and interesting twist to an old and fun idea. Okay, I'll get off my soap box.



Animating with Object-Oriented SpaceFoils

The true beauty of object-oriented programming is the reusability of code. I use the same code to render the SpaceFoil as I use to render the opposing ship and

both ships' missiles. The trick is to get one object set up correctly as a polygon abstraction, and from there on, you just cut and paste code. What I did for SpaceFoil was set up an array of 32 objects. This was overkill, because I actually used only 12 objects in the array. These objects were the SpaceFoil, two sets of afterburner flames for both ships, plus a stowed dummy torpedo and a live torpedo for each ship. The dummy torpedo is a non-visible torpedo carried along with the ships, which moves along with the same speed and rotation angle as its ship. Using the wonders of the plus-plus in C++, the Shape object class has an overloaded assignment operator. So when a player fires a missile, the Shape object grabs the current instantiation of the dummy missile and initializes the missile object to these values. When the missile is fired it is pointing in the same angle and has the same momentum as the ship that fired it. The benefit is that no computation was required to determine all this for the new missile to be fired.

So how can the Shape class be used to create and move a SpaceFoil? Well, start with an array of points. The array for the SpaceFoil points is shown in Listing 6-28.

LISTING 6-28 *sfArray* point list

```
Point sfArray[] =
{
  {-1,-4},  {-4, -2},  {-6,0},    {-16,4},
  {-14,7},  {-6,5},    {-4,6},    {-2,3},
  {2,3},    {4,6},     {6,5},     {14,7},
  {16,4},   {6,0},     {4,-2},    {1,-4}
};
```

These coordinates seem meaningless at first glance, but they make sense when you plot the coordinates on a piece of graph paper. No, they don't form the shape of the Grimace or the Hamburgler like on the back of a MacDonald's Happy Meal bag; this is the polygon definition of our ship, see Figure 6-39. The polygon shape of the ship is said to be clockwise, because this is the manner in which to read the vertices. The first vertex is at point (-1,-4) in the array; the points in the array correspond to the points in the figure proceeding in a clockwise manner as points are read.

Initializing an Array of Objects you could simply define a Shape object individually for every polygon you wish to use. For example, you could declare,

```
Shape* sSpaceFoil, sRightFlame, sLeftFlame, sTorpedoDummy, sTorpedo;
```

which is perfectly fine, except this could optimize not just the code, but other cut-and-pasting time, by declaring an array of shape objects using the following sample code in a header file, as shown in Listing 6-29.

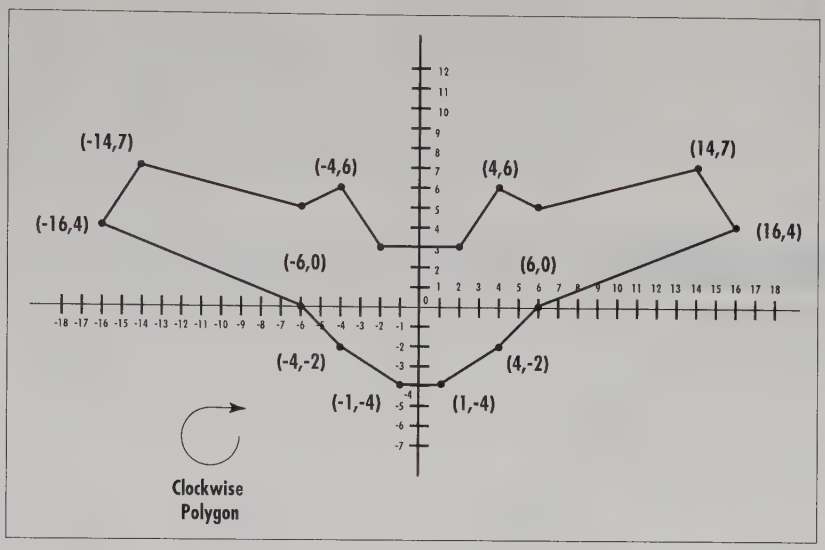


FIGURE 6-39

List of vertices defining the SpaceFoil

LISTING 6-29 An enumerated list of game shapes

```
#define MAX_OBJECT_ENTRIES 32

enum shapeIndex { SPACE_FOIL_INDEX,
                  ENEMY_INDEX,
                  SF_RIGHT_FLAME_INDEX,
                  SF_LEFT_FLAME_INDEX,
                  ENEMY_RIGHT_FLAME_INDEX,
                  ENEMY_LEFT_FLAME_INDEX,
                  SF_TORPEDO_INDEX,
                  SF_TORPEDO_VOLLEY_INDEX,
                  ENEMY_DISRUPTOR_BOLT_INDEX,
                  ENEMY_DISRUPTOR_INDEX,
                  SF_EXPLOSION_INDEX,
                  ENEMY_EXPLOSION_INDEX
                };
```

The following object declaration must appear in the source code,

```
Shape *object[MAX_OBJECT_ENTRIES];
```

Now there is a declaration of 32 objects that can be used at will. To add a shape definition, simply add an index to it in the enum shapeIndex list. In this is the only place to modify the code in the header file, when adding a shape. If the index is less than 32, there is already a Shape object reserved for it. In this case, I only used 12 of the 32 objects. When going back to clean up the code, simply change the MAX_OBJECT_ENTRIES value to 12. I think the light bulb might be starting to flicker for you here. You're trying to keep a majority of the code changes where? In the header file!

At this point you have only declared the objects. Now the array of objects must be initialized before actually using them. This is trivial, but don't tell your manager; it's important to keep your manager starry eyed with the illusion you're a miracle-worker. This is a little piece of wisdom I picked up from Chief Engineer Montgomery Scott on the old *Star Trek* episodes. To initialize the array of 32 objects use the code in Listing 6-30.

LISTING 6-30 Initializing a shape object array

```
void initShapes()
{
  int index;

  for (index = 0; index < MAX_OBJECT_ENTRIES; index++)
    object[index] = new Shape;
}
```

A little too easy wasn't it? Again, notice that all modifications for this code are from the header file. If I want to increase or decrease the number of objects in my code, I do so by changing the value of MAX_OBJECT_ENTRIES in the header. Never hard code any values in the source code files. This can be a nightmare when it comes to serious code modifications in a project.

When the array of objects has been initialized, start using the objects. Recall, in the header file, I have enumerated indices into my array object. This means that I can arbitrarily use any object in the array that's within the index range of zero to 31. Having added SPACE_FOIL_INDEX to the enumerated list, I can initialize my SpaceFoil object; see Listing 6-31.

LISTING 6-31 Initializing the spaceship Shape objects

```
void initSpaceFoil()
{
  int objectSize;

  objectSize = sizeof(sfArray)/sizeof(sfArray[0]);
  initObject(object[SPACE_FOIL_INDEX],sfArray,objectSize,320,240);
);
```

The function initSpaceFoil() analyzes the size of the Point array, *sfArray*, which contains the vertices of the SpaceFoil polygon. Following this computation it passes the object, the Point array, the array size, and the initialization screen coordinates to the function initObject(). The code for function initObject() is shown in Listing 6-32.

LISTING 6-32 Initializing a Shape object

```
void initObject( Shape* s,
                 Point *pointArray,
                 int size,
```

```
                double x,
                double y)
{
  s->encodeObject( pointArray, size );
  s->setAngle(0);   // initial display angle
  s->setVector(0,0);  // initialize V = 0
  s->setScreenXY(x,y);  // default = center
}
```

A pointer to the Shape object is passed to initObject(). Here the underlying code is actually a bundling of class Shape member functions to initialize the Shape object. The member function Shape::encodeObject() is where to actually read the Point array and assign its values to the shape definition array, *shapeDef*, in the object. The other embedded member functions initialize the screen attributes. The member function Shape::setAngle() initializes the rotation angle to zero. Likewise, the velocity vector data members are initialized to zero; the object is not moving. Finally, assign the position where to display the shape on the screen. This function provides a generic mechanism to cut and paste initialization code for the rest of your shape objects. For instance to initialize the SpaceFoil's afterburner flames, this code can be used, assuming there are Point arrays, *sfRightFlame*, and *sfLeftFlame*, for the right and left flames; see Listing 6-33.

LISTING 6-33 Initializing the afterburner Shape objects

```
void initSpaceFoilFlames()
{
  int size;
  int objectSize;

  size = sizeof(sfRightFlame[0]);
  objectSize = sizeof(sfRightFlame)/size;
  initObject( object[SF_RIGHT_FLAME_INDEX],
                sfRightFlame, objectSize, 320, 240 );
  size = sizeof(sfLeftFlame[0]);
  objectSize = sizeof(sfLeftFlame)/size;
  initObject( object[SF_LEFT_FLAME_INDEX],
                sfLeftFlame, objectSize, 320, 240 );
}
```

Again, the only requirements for cutting and pasting are that we added the indices, SF_RIGHT_FLAME_INDEX and SF_LEFT_FLAME_INDEX, to the enumerated list in the header file, and provided polygon definitions in Point arrays in the source code. Now you can cut-and-paste to your heart's content. Proceed in this fashion to initialize the enemy ship, the enemy ship's afterburners, both ship's missiles and their explosions. A sad but true fact is that in the eyes of your manager you are paid to produce and not to create. That's why you learn to cut-and-paste.

SpaceFoil Navigation 101

Initializing the objects only provides enough information to statically display the shapes on the game screen. To provide a means to turn and propel the SpaceFoil, essentially, the objects need to be rotated right and left, as well as updating the velocity vectors for screen motion. The rotation event is caused by determining if a right or left arrow key was pressed. To rotate the SpaceFoil to the left, call the function rotateSpaceFoilLeft(), shown in Listing 6-34.

LISTING 6-34 The rotateSpaceFoilLeft() function

```
void rotateSpaceFoilLeft()
{
  rotateObjectLeft( object[SPACE_FOIL_INDEX] );
  rotateObjectLeft( object[SF_RIGHT_FLAME_INDEX] );
  rotateObjectLeft( object[SF_LEFT_FLAME_INDEX] );
  rotateObjectLeft( object[SF_TORPEDO_INDEX] );
}
```

Cutting and pasting now occurs again in the body of the code. The function rotateSpaceFoilLeft() updates the SpaceFoil direction when it rotates left. Notice that the afterburner flames are updated and so is the dummy missile. All the Shape objects use the function rotateObjectLeft() to handle the rotation to the left. This function is shown in Listing 6-35.

LISTING 6-35 The rotateObjectLeft() function

```
void rotateObjectLeft( Shape *s )
{
  s->stepAngle( -ROTATE_STEP );
  if (s->getShapeAngle() < 0)
    s->setAngle(360 - ROTATE_STEP );
}
```

Key points here are that rotations to the left are brought about by setting the object with a negative step angle. Note also that the angle is a protected data member. This means you can't access the value of the angle in the code. You must request to see a copy of it by using the Shape class member function getShapeAngle(). This is called data-hiding. Why hide data? This is done for the same reason code changes on the header files are emphasized. It provides a mechanism to minimize code changes when the code needs to be altered. In some cases data hiding assists in hiding our booboos when releasing a fix.

To rotate the Shape object to the right, use a similar piece of code. Its differences reflect the fact that the rotation is now to the right; that is, in a positive angular direction. To rotate a Shape to the right use the function rotateObjectRight(), shown in Listing 6-36.

LISTING 6-36 The rotateObjectRight() function

```
void rotateObjectRight( Shape *s )
{
  s->stepAngle( ROTATE_STEP );
  if (s->getShapeAngle() > 359)
    s->setAngle( 0 + ROTATE_STEP );
}
```

The functions rotateObjectLeft() and rotateObjectRight() have not performed any rotations. They are actually functions to ensure that we set the proper angle we wish to rotate with. The rotation is actually performed with the function objectTransformation(), shown in Listing 6-37.

LISTING 6-37 Rotating an object with objectTransformation()

```
void objectTransformation( Shape *s )
{
  s->compositeTransformation();
}
```

This is the call that rotates the Shape object. It is simply a repackaged rendition of the Shape class member function Shape::compositeTransformation(); see Listing 6-38.

LISTING 6-38 The Shape member function compositeTransformation()

```
void Shape::compositeTransformation()
{
  int index;

  rotateShape( angle );

  for (index = 0; index < pointEntries; index++)
  {
    sprite[index].x += screen.x;
      sprite[index].y += screen.y;
  }
}
```

The member function actually handles the rotation of the polygon. Remember the earlier discussion for data hiding? The member function Shape::rotateShape() can directly access the data member angle with having to request a copy of it. What has been done by rebundling the code is that in the event a bug shows up in the Shape class code, I only have to fix and re-compile my class Shape code. The development group that was working on the SpaceFoil rotation code only needs a copy of my new object code. They merely relink their code to my new object code and we're both up and running, and I still have a job!



Bring Us Around 1/3 Impulse, Mr. Sulu

Now rotate the ship on the screen. But setting dead center of the screen really isn't all that safe. You know from what you've learned watching *Star Trek* that the Klingons never take prisoners. So you'll need to provide some maneuverability with the SpaceFoil. Motion is provided for the SpaceFoil by the function call thruster(); see Listing 6-39.

LISTING 6-39 Updating the SpaceFoil velocity components

```
void thruster( Shape* s )
{
  double radianAngle;
  double vx, vy;

  // update Space Foil
  radianAngle = s->getShapeAngle() * RADIAN_UNIT;

  vx = s->getvx() + sin(radianAngle)*SPEED;
  vy = s->getvy() - cos(radianAngle)*SPEED;
  s->setVector(vx,vy);
}
```

As you can see, the order of operation of maneuvering the SpaceFoil is that you must perform rotations first, then update the velocity vectors in the desired directions of travel. Using trigonometric relations the *x* component of velocity is

```
vx  = vx + vx*sin(radianAngle) * velocity constant
```

The *y* component for velocity is computed in a similar manner,

```
vy  = vy - cos(radianAngle) * velocity constant
```

The velocity constant is an arbitrary value to reduce the speed of the SpaceFoil on the screen. Without this the SpaceFoil will warp across your screen. This is not Wolf359 and you're not engaging the Borg, so all battles will be fought at impulse speeds. To bring the impulse engines online use the code in Listing 6-40.

LISTING 6-40 Activating the SpaceFoil engines

```
void impulseSpaceFoil()
{
  bShowFlame = TRUE;  // display the flames

  // update Space Foil
  thruster(object[SPACE_FOIL_INDEX]);

  // update Space Foil right flame
```

```
  thruster(object[SF_RIGHT_FLAME_INDEX]);

  // update Space Foil left flame
  thruster(object[SF_LEFT_FLAME_INDEX]);

  // update the stowed torpedo
thruster(object[SF_TORPEDO_INDEX]);
}
```

As you can see, I have reduced you to a Kelly Girl *temp admin* by using object-oriented design techniques. But that's all right; just place your college degree in location where your managers are forced to look at it every time they enter your cube. Besides, OOP is a managerial buzzword. You'll be the department programming *guru* (that's a buzzword that needs to have a stake driven through its heart!) in your managers' eyes. Didn't you ever wonder what they do at those all day managerial offsites? They're taught new buzzwords.

All the screen and vector components of the SpaceFoil have been changed. You have performed a rotation and a change in the velocity vector components. You need to signal the object to update these values before you render it to the screen. You do this with a function call to moveObject(), shown in Listing 6-41.

LISTING 6-41 Moving a Shape object on the screen

```
void moveObject( Shape *s )
{
  if (s != NULL)
   s->moveShape();
}
```

The function moveObject() is used to call the Shape class function member, Shape::moveShape(); this wrapping is merely for the convenience and minimization of effort during the cut-and-pasting process. The function member Shape::moveShape() is shown in Listing 6-42.

LISTING 6-42 The Shape member function moveShape()

```
#define SCREEN_MAX_X 640
#define SCREEN_MIN_X 0
#define SCREEN_MAX_Y 480
#define SCREEN_MIN_Y 0

void Shape::moveShape()
{

  screen.x += vector.x;
  screen.y += vector.y;

  // handle screen wrap around
```

continued on next page

continued from previous page

```
  if (screen.x < SCREEN_MIN_X)
    screen.x = SCREEN_MAX_X;
  if (screen.x > SCREEN_MAX_X)
    screen.x = SCREEN_MIN_X;

  if (screen.y < SCREEN_MIN_Y)
    screen.y = 479;
  if (screen.y > 479)
    screen.y = SCREEN_MIN_Y;
}
```

The SpaceFoil shape is now being moved to its logical new location before being rendered to the screen. This member function takes the velocity components and updates the screen location's *x* and *y* values. These values are then checked against the game screen bounds. If you go off the screen, readjust the coordinates to appear on the other side of the screen from where you want off. This is called screen wrapping. (For those of you who take your space battles very seriously, it's also known as the *Picard Maneuver*.) It's an effect used in games like *Asteroids*, the old vector graphics *Star Trek* game, *SpaceWar*, and others.



Engage!

All this wonderful excitement is taking place and the players haven't seen a pixel of it. We need to draw the updates to the screen. Figure 6-40 gives us a glimpse at what our shrink-wrapped end results might look like. The Shape class provides the member function Shape::renderShape() to draw the object to the screen. The code for this is shown in Listing 6-43.

LISTING 6-43 The Shape member function renderShape()

```
void Shape::renderShape()
{
  int index;

  for (index = 1; index < pointEntries; index++)
  {
    BresenhamLine(
              (int)(sprite[index - 1].x),
              (int)(sprite[index - 1].y),
              (int)(sprite[index].x),
              (int)(sprite[index].y),
              127);

  }
```

```
  // join the last vertex to the first in the sprite Point array
  BresenhamLine(
           (int)(sprite[index - 1].x),
           (int)(sprite[index - 1].y),
           (int)(sprite[0].x),
           (int)(sprite[0].y),
           127);
}
```

To draw an object to the screen we can call the Shape class member function directly, shown in Listing 6-44.

LISTING 6-44 Drawing the SpaceFoil explosion

```
void renderExplosion()
{
   // draw the enemies green bleeding flaming guts
   if (bEnemyCoreBreach == TRUE)
     object[ENEMY_EXPLOSION_INDEX]->renderShape();
   // activate the emergency transporter signature
   if ( bSpaceFoilCoreBreach == TRUE )
     object[SF_EXPLOSION_INDEX]->renderShape();
}
```

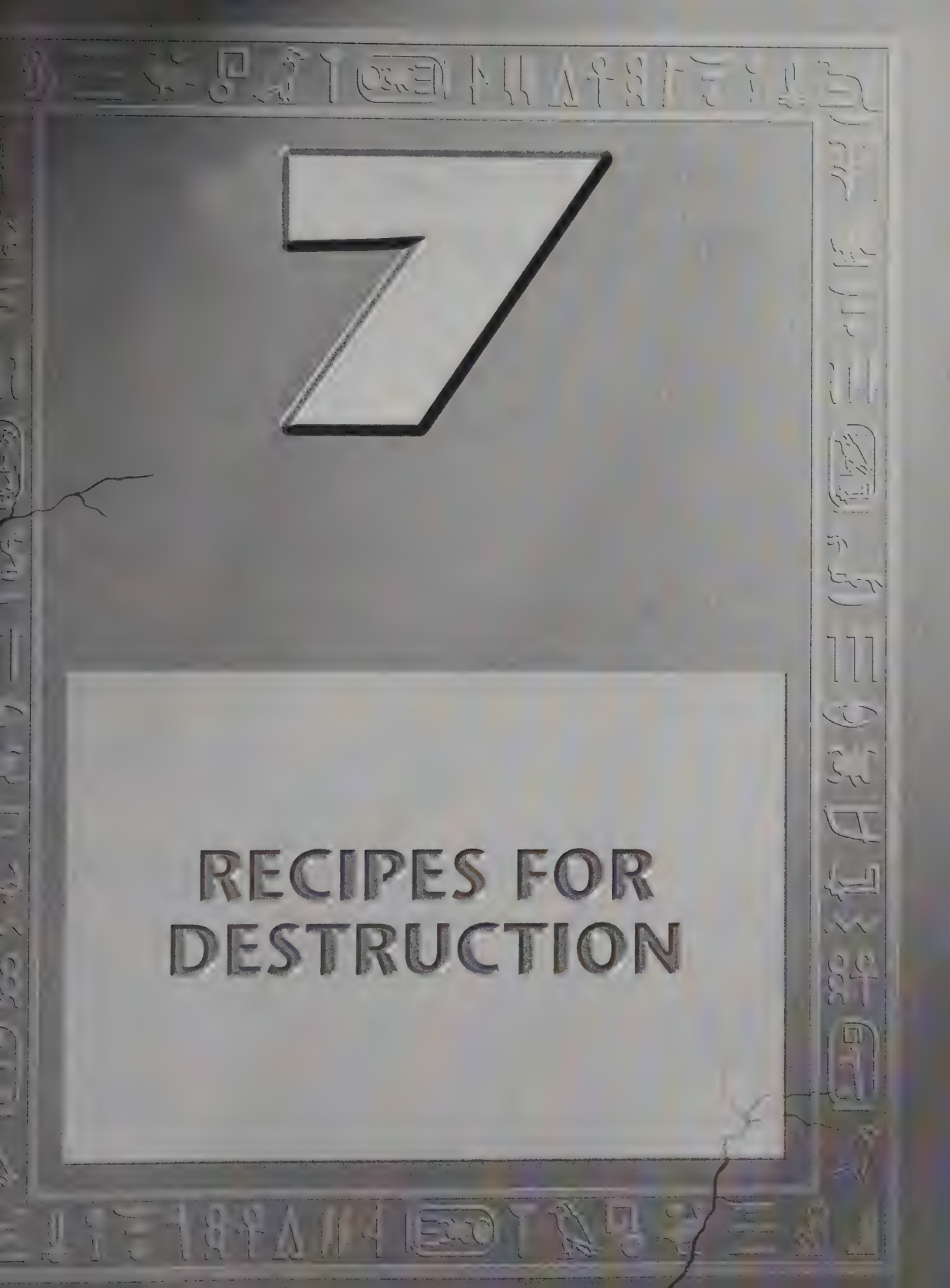


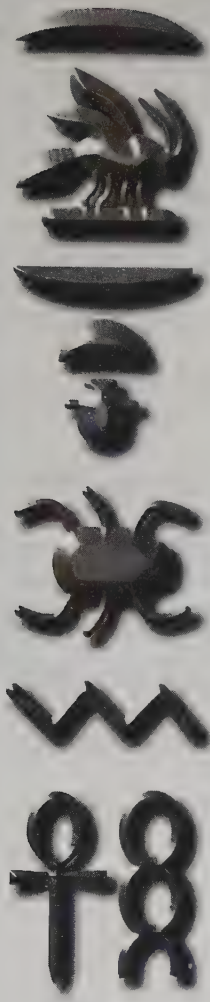
FIGURE 6-40
A screenshot of the multiple polygons used in SpaceFoil



Conclusion: Polygons and Saving the Universe

A lot of material was covered in this chapter. You learned how to plot a pixel using a DirectDrawSurface and you built on this foundation to construct object-oriented polygon spaceships. You may want to spend some time and play around with the *SpaceFoil* game. Give it photon torpedoes and hyperspace. Or you can tear it apart and make your own game. It's kind of like a Mr. Potato Head for software developers. You can snap on what you want in order to create your own game. The concepts presented here will be used in Chapter 8 to build a vector graphics style *Asteroids* game using DirectDraw.


7
RECIPES FOR
DESTRUCTION



RECIPES FOR DESTRUCTION

Underneath the game sounds and graphics, your game application is composed of algorithms and data structures. When you remove the texture mappings and the sprites from *DOOM*, you have linked lists and dynamic structures of memory to manipulate the game enemies. In this chapter, you'll examine the data structures such as linked lists and binary trees that are commonly used in game programming. In the second half of this chapter, you'll look at how to use concurrent programming techniques in game design using the multitasking features of the Windows 95 operating system.

EPISODE NOT MY MUMMY



"Bary, hold up the map you got so I can see it," 13 asked. "I thought so."

"Thought what?" I replied.

"That game map is a circuit diagram to the artifact," 13 exclaimed with self-assurance.

"Are you abusing any chemical substances I should be aware of?" I teased.

"No. Kala left to go back and get some of her notes. And I assure you, I am not taking any drugs. Take a look at the room with the gold coiled serpents," 13 replied seriously.

"Yeah, some expensive snakes. So what?" I didn't know where this kid came up with these things. But I did my best to humor him.

"Look at the room going into the serpent chamber. There are two parallel passages leading into a bell-shaped room. One passage leads out from the other end of the room. This looks like a primitive NAND gate. Look at the passage parallel to yours. It comes out of a similar NAND gate. Bary, you're standing in a set/reset flip-flop!"

"Get out of Dodge! That's impossible. 13, I think the Grinch must have fed you paint chips in your baby formula."

"Bary, I bet when you enter the serpent's chamber there will be huge wire toroids of gold or copper wire."

"What are you getting at, 13?" He had me pretty confused.

"You're not thinking like a several-thousand-year-old sorcerer, Bary. There was no Intel. There were no Pacific rim memory manufacturers. The guy only had his twisted evil brains, gold, and copper. Our first computers used core memory. These were strands of copper wire wrapped in a toroid. When the toroids experienced a torque, their memory was set. Agamemnon is using the same thing," 13 cleverly concluded.

"So, if I go through this passage, or flip-flop, I'll be in core memory?" I tried to piece the thing together myself.

"Yes, Bary. Go there and let's see if we can shut down some critical memory."

Just as I made my way to the door of the core memory chamber, a golden crossbow materialized in front of me with two arrow quivers lying next to it.

"Oh boy," I muttered.

"Hey Bary, don't grab the crossbow. Remember what happened in the last chamber?"

He didn't have to remind me. I had the same terrible thought in my head. Unfortunately, there was no going back either. The core memory was one of our objectives. I had to press on. I took a deep breath and grabbed the crossbow and quivers. The rumbling noise of a stone door lifting reverberated through the chamber. Then the lights went out. I deduced that standing in the middle of the room wasn't the most intelligent thing to do. I could see light from the chamber on the other side, but I also saw long shadows of slow moving *things* heading my way. Guessing by the shadows, there were approximately four.

The crossbow already appeared to be loaded. Which was a good thing because I had never used one of these before. As the first long shadow moved into view of the door, I could see it was a mummy with its arms reaching out, making a moaning noise.

With the crossbow, I took aim on the wrapped zombie and fired. The room illuminated in a yellowish gold light as the magic arrow took flight. I squeezed off about five more shots before the first mummy vaporized into thin air. The second and third mummy fell to the same demise as the first.

This is easy, I thought to myself. Then I saw the shadow of the final mummy come into my view. This shadow seemed to be different. It was carrying something I couldn't make out. When it finally came into view, I saw that it too

had a crossbow. Diving out of the way, I took cover behind a large column. If I waited a second longer, I would have fallen to its first shot. The mummy continued mindlessly firing at where I had stood. It let out a grueling moan. Taking cover from behind my column, I released four shots and ran for cover behind another pillar. It groaned as my four shots hit, but it kept coming. Four shots were lethal to the other three. I fired off five shots, all direct hits, and the monster moaned in agony. But it kept coming. My first quiver was empty and it vanished. My second quiver loaded more arrows into the crossbow. Taking aim, I sank five more shots into the mummy and it finally vaporized. The light came back into my room and new quivers of arrows lay where the mummies had fallen.

"Okay 13, let's see what lies in store for us in core memory. If you see any secret chambers, let me know. I would feel a lot better with a rocket launcher or an auto cannon than a crossbow and pouch of magic."



The Linked List

The linked list is one of the most flexible data structures used in game applications. The list data structure contains its information in a linear order. An array of structures is similar to a link list, but the implementations differ significantly. An array is linear ordered by array indices where the link list is ordered by pointers to each object in the list. Lists are extremely efficient data structures in that they can shrink and grow as the application requires it. In comparison, an array data structure requires a predetermined size at run-time of the application. The size of the array is fixed throughout the execution of the program.

The classical linked list defined in text books, such as *Data Structures and Algorithms*, by Aho, Hopcraft, and Ullman, is an ordered finite sequence of zero or more elements. A list can have elements consisting of integers, real numbers, lists of data structures, and so on. The mathematical notation for a list is to write the elements of the list, separated by commas, inside closing parentheses, like this:

$L = (a_1, a_2, a_3, a_4, ..., a_n)$

Lists used by game developers typically have sprites, tiles, and other game objects as their list type. In Chapter 8, you will use a list to maintain space debris for an Asteroids game.



The List Data Structure

In the C language, you implement a list by declaring a structure called a cell or a node. In this discussion, we will call it a node. Each node consists of an

element, such as an integer, a pointer to a sprite, or a tile structure. A node also has a pointer to the next node in the linked list. The structure declaration for a linked list, in C, is shown in Listing 7-1.

LISTING 7-1 Classic C declaration for a linked list

```
struct node {
  int value;
  struct node *next;
};

typedef struct node *list;
```

You have declared a pointer, *list*, to reference a list of node structures. The next field of each node structure points to the next node structure in the list. Each node in your list contains an integer element. Figure 7-1 shows your list using the abstraction a_1, a_2, ..., a_n to label the nodes in the linked list. The last node in the list, a_n, has its next field set to NULL. This identifies the end of the list. Our pointer, *list*, represented by the *listPtr* in the figure, is the pointer to the *head* of the list.

The list is a mathematical model, called an abstraction, as you have seen with the list notation. The linked list is a data structure to implement the list model.



Object-Oriented Lists

Operations that may be performed on a set of objects, such as inserting, deleting, and searching, are called abstract data types (ADTs). Using C++, you can take a collection of operations, such as ADTs, and form a class. That is, you can create a list object that will allow you to insert, search, and remove node objects from a list object.



Designing the cList Class

The list object has some fundamental operations you can perform on it. You need to be able to insert into the list, as well as search and remove from the list.

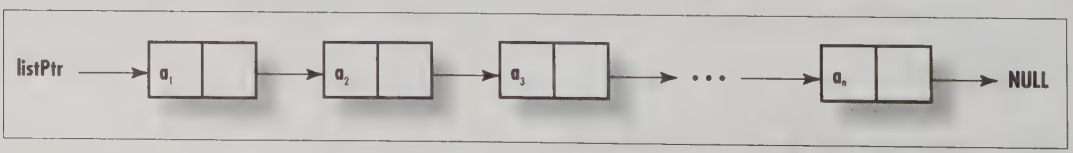


FIGURE 7-1 Classic representation of the linked list

The list itself has a set of states that can exist during its operation. A list may be empty, where there are no elements in the list. A list may also be full, in which case you have an overflow, which is very serious. In C++, the list is full when the program's free store is exhausted (has no more memory to give out).

If you have never dealt with object-oriented design before, don't worry. This is a hands-on example. You'll go through the details of building a list object and later in the book, you'll modify this object to fit your game application's needs. Begin your object-oriented list design by writing down the operations of the list on a sheet of paper. This mock class design is shown in Listing 7-2.

LISTING 7-2 Design for a C++ list class

```
class cList {
  public:
    int element;
    insert( int element );
    remove( int element );
    BOOL search( int element );
};
```

I have left out the constructors and destructors to simplify the example. This is just a design listing, or a prototype wish sheet, if you will, of what you want your list object to do. This doesn't closely resemble the classical linked list yet. Your next step in the design process is to design the node object and the list object. In the classical example, the list abstraction was a list of nodes, successively linked by next pointers. In the object-oriented implementation, the node is an object and the list is an object, as shown in Figure 7-2. A list object is the head of the list, which references an empty list or a list with some node objects in it. The node class contains the elements and the link to the next node object. The C++ node class declaration is shown in Listing 7-3.

LISTING 7-3 Declaration of a C++ node class

```
class cListNode {
friend class cList;
private:
  int element;
  cListNode *next;

  cListNode( int i, cListNode *node = NULL) { element = i; next = node; }
};
```

The class cListNode consists of two data types: an integer data element, and a pointer to the next cListNode object. You need the list class in order to build your linked-list of cListNode objects. The declaration for the C++ cList class is shown in Listing 7-4.

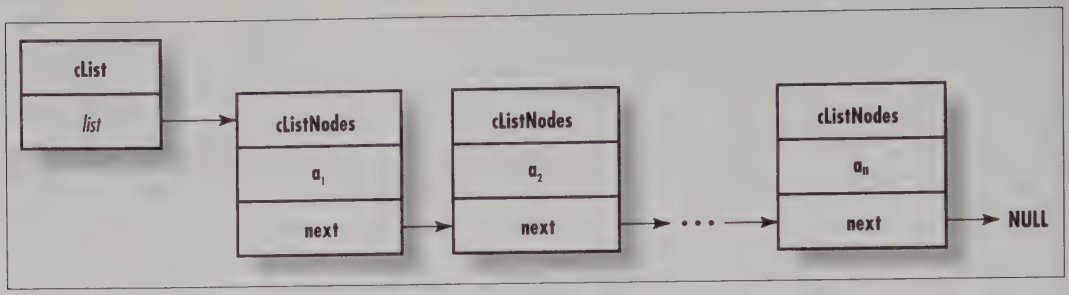


FIGURE 7-2 Object-oriented list

LISTING 7-4 Declaration for a C++ cList class

```
class cList {
private:
  cListNode *list;

public:
  cList() { list = NULL; }
  ~cList();

  void InsertList( int i );
  void DeleteList();
};
```

The default constructor in class cList always initializes the list to EMPTY or NULL. Recall from Chapter 2 that a constructor is a special member function that initializes the class. The constructor is given the name of the class. Also, the special member function with the tilde, '~', preceding the class name, is the destructor, which is responsible for deallocation of the object. To use your list object in your code, you make the following declaration:

```
cList l;
```

When this line of code is encountered in your program, the class constructor is immediately called and the list object *l* is initialized to NULL. You can now begin to implement the operations of insert, remove, and search into cList class. Before doing so, it would make your work a lot easier to declare a cListNode object tail. This does nothing more than serve as a marker to represent the last node in your list object. (See Figure 7-3.) The modifications to the cList class declaration can be found in Listing 7-5.

LISTING 7-5 Adding a head and a tail object to the cList class

```
class cList {
private:
  cListNode *list;
  cListNode *tail;
```

```
public:
  cList() { list = NULL; tail = NULL; }
  ~cList();
    :
};
```

Inserting a Node Object into a List

Notice in your class constructor, you have initialized two list object pointers to NULL. These are the pointer *list* and the last cListNode object pointer, *tail*. This provides you with enough information to begin inserting to the list. The code to insert integer elements at the front of the list can be found in Listing 7-6.

LISTING 7-6 Inserting into the cList object

```
void cList::InsertList( int i )
{
  cListNode *tempNode = new cListNode( i, list );
  assert(tempNode != 0);

  if (list == NULL)
    tail = tempNode;

  list = tempNode;
}
```

The class member function InsertList() initializes a temporary node object, *tempNode*. The C++ new operator allocates the memory from the program's free store. The class cListNode constructor is called and passes the constructor arguments. The first argument is the actual data element for the node, and the second is the list object pointer. The conditional test in the code checks to see if this is the first element you are inserting into the list. If this is the first element, both the list and the tail object pointer reference the same node object, and the list is EMPTY, as shown in Figure 7-4.

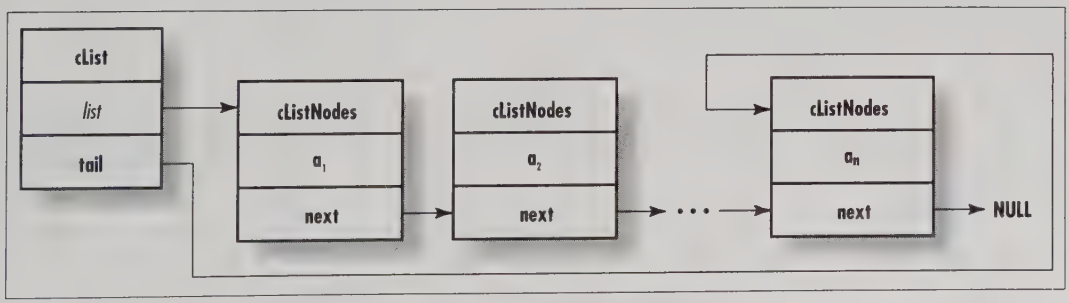


FIGURE 7-3 Using a tail object to track the end of a list object

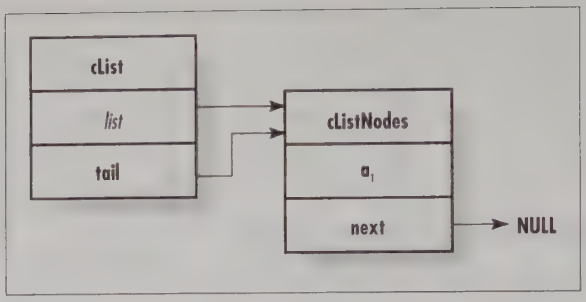


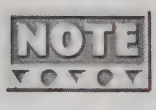
FIGURE 7-4
Inserting the first element in a list object

Your head may be spinning around with all this new object stuff, so let's walk through what the InsertList code is doing. To insert a new element into the list object, you must first create a node object to contain your element and its pointer to the next node object in the list. (See Listing 7-7.)

LISTING 7-7 Creating a node object

```
cListNode *tempNode = new cListNode( element, list );
assert(tempNode != 0);
```

You have created a *tempNode* using the cListNode constructor with the new operator. The *tempNode* is initialized to contain the value you wish to insert into the list object. You haven't, as of yet, inserted the node object into your list object. Figure 7-5 illustrates what has taken place with this line of code.



If the cListNode object was not allocated because the program's free store was exhausted, the program will terminate. The code to do this is as follows:

```
assert(tempNode != 0);
```

Remember, you must always make sure you get the memory you requested to be allocated to the program.

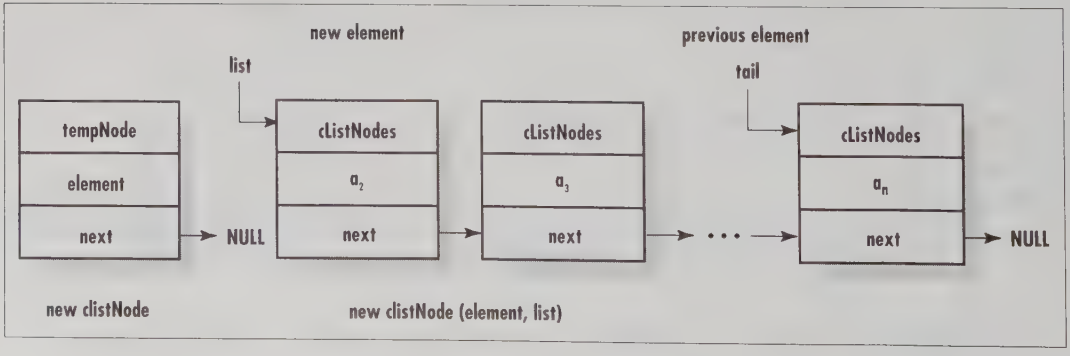


FIGURE 7-5 Creating a new cListNode object in InsertList()

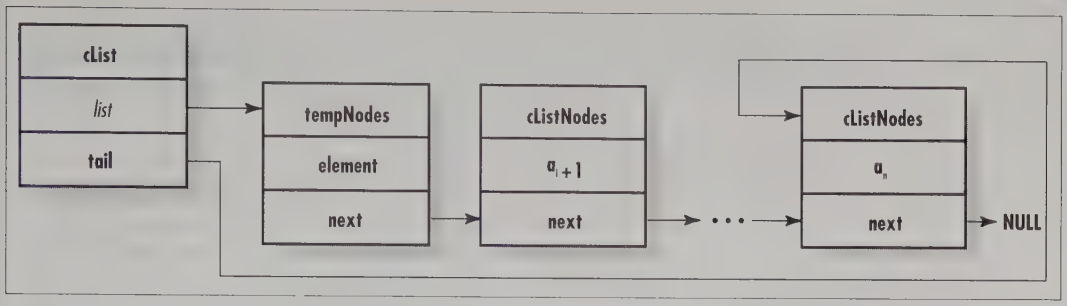


FIGURE 7-6 Assign the list object pointer to reference the new node object

If the memory was allocated from the program's free store successfully, all you have left to do is to assign the list object pointer *list* to reference the new node. This is shown in Figure 7-6.

That's all there is to it. As I mentioned earlier, object-oriented programming is very different from structured programming. Learning to program using C++ is a lot like learning calculus. At first it doesn't make an ounce of sense and then the more you apply it, the more you understand what's going on.



Removing a Node Object from the List

The list object is capable of growing and shrinking. This concept is really not all that confusing because the cases for the list object are identical to that of the list ADT. To remove an element from the list, you have four possible cases of what might occur. The list may be empty, so the search will terminate. A trivial case may be that the element is in the front node of the list, minimizing the search time. The following two cases require transversing the list object. The element may be in the list somewhere, or it may not be in the list at all. You may also have multiple items of the same element in the list. This would require multiple removals.

Consider the two most trivial cases: the list is empty, or the element is at the front of the list. The first case is simple; the list is *empty*. You want to make sure you have a list before you proceed with a search. Use the code in Listing 7-8, RemoveList, to test for the existence of a list object.

LISTING 7-8 Checking to see if an element is in the list

```
cListNode *n = list;
 int elementFound = 0;

 // trivial case: the list is EMPTY
 if ((list == n) == NULL)
 {
   tail = NULL;
   return elementFound;
 }
```

Here the code returns 0, indicating no occurrence of the element was found. The list object is empty. In the event one or multiple occurrences are located, the function will return a count of how many it found. This was a little trick I picked up from Stanley Lippman's book, *C++ Primer*. Lippman's book is *the* book on C++ programming conventions.

In your second trivial case, you see if the element is at the head of the list. To handle this case, you simply create a temporary cListNode object and set this object to reference the next node object in the list. This is kind of like your bookmark in the list. You want to delete the first node object in the list, but you need a placeholder to reference the rest of the list. With your virtual bookmark in place you can nuke your first node object pointer. You then set what was the second node object in the list as the first node object in the list. Perhaps a picture can explain a thousand words. (See Figure 7-7.) The code for removing the first element in a list can be found in Listing 7-9.

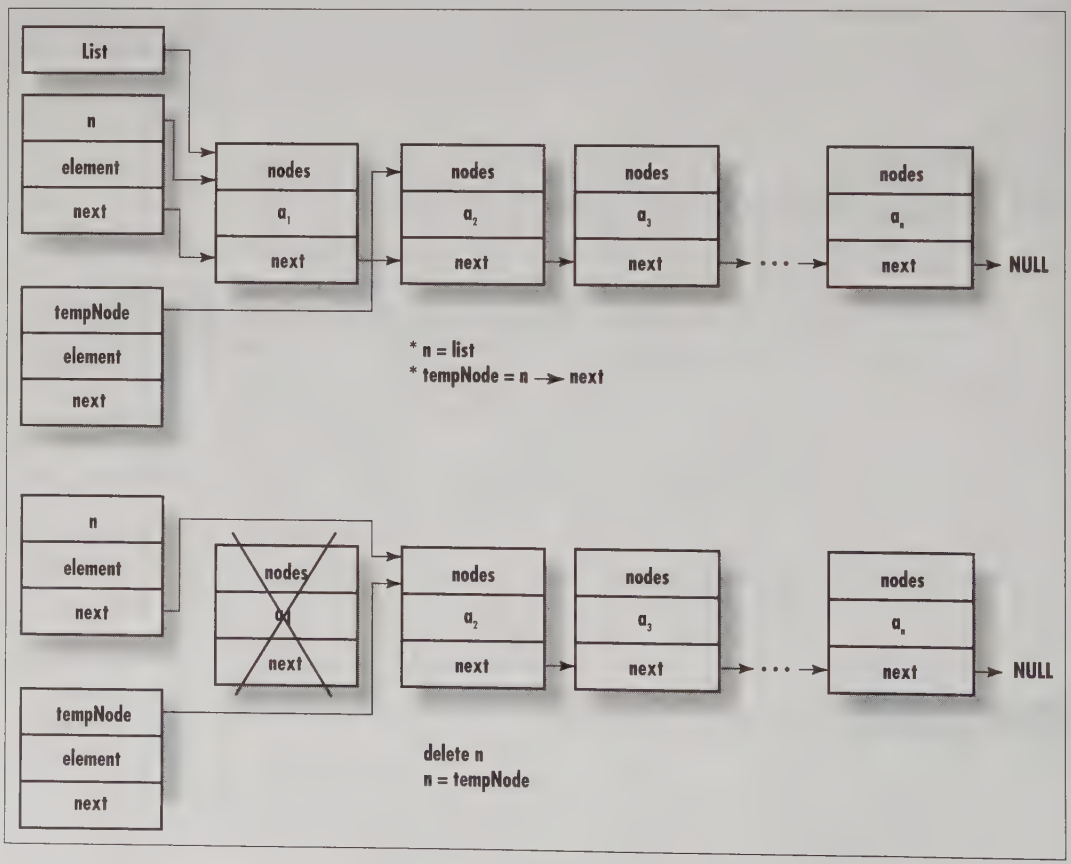


FIGURE 7-7 Removing the first node object in a list

LISTING 7-9 Removing the first item in the cList object

```
// trivial case: element is the first in list
  while( n && n->element == aNumber)
  {
    cListNode *tempNode = n->next;  // get pointer to next node
    delete n;
    n = tempNode;
    elementFound++;       // element deleted
  }
```

Now you have the not-so-trivial cases to deal with. You know that the list object is not empty and that the element you are searching for is not in the first node in the list. You have several possibilities to contend with. One, the element is not in the list at all, forcing you to transverse the entire list. Second, the element is somewhere in the list and there may be the possibility of multiple occurrences. Any of the above plausible cases is handled with the next snippet of code shown in Listing 7-10.

LISTING 7-10 Searching and removing items from the cList object

```
  cListNode *prev = n;
  n = n->next;

  // trivial cases failed. search for the element
  while( n != NULL )
  {
    // element found?
    if (n->element == aNumber)
    {
    prev->next = n->next;
    if (tail == n)
       tail = prev;
    delete n;
    n = prev->next;
    elementFound++;
  }
  else // not found, go to next node
  {
    prev = n;
     n = n->next;
    }
}
return elementFound;
```

By the time you have arrived to this piece of code, you know that the list is *non-empty*. Therefore, you may initialize your pointers to commence a node-to-node search for the vile element. It can't hide from you! You initialize your pointers by creating a previous node pointer object, which is always one step

behind in the search process as you search each node. To set up the pointers, use this code before you enter the while loop to search:

```
cListNode *prev = n;
  n = n->next;
```

Figure 7-8 helps clarify what is going on. Two situations arise. One, you haven't found what you are looking for and you continue to search the list, or, second, you find the node object with the element you are searching for. In the case where the element is not found, you progress with your search within the loop shown in Listing 7-11.

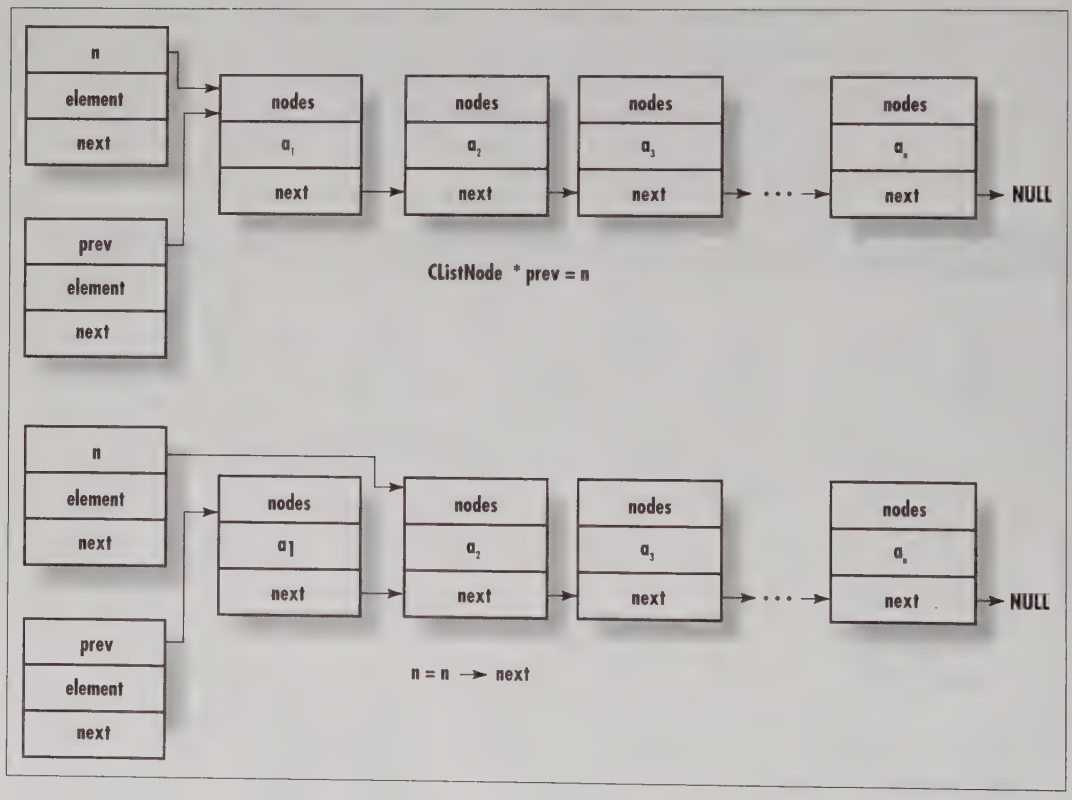


FIGURE 7-8 Searching a list of node objects

LISTING 7-11 Searching for an item in the list

```
while()
  {
    // element found?
    if (n->element == aNumber)
    {
       :
    }
    else // not found, go to next node
    {
      prev = n;
      n = n->next;
    }
  }
}
```

From Figure 7-9, you see that the operation is nearly identical to that of initializing the pointers for the search. You assign the node object *prev* to reference the current node object *n*, and you advance the node object pointer *n* to the next node. In the event you find the element in the list you were looking for, two possibilities may arise. Case number one, the element was found at the end of the list, and case number two, the element was found in the list. In either event, the code for handling an instance found is shown in Listing 7-12.

LISTING 7-12 Deleting the item once it's found

```
    if (n->element == aNumber)
    {
       prev->next = n->next;
       if (tail == n)
         tail = prev;
       delete n;
       n = prev->next;
       elementFound++;
    }
    else // not found, go to next node
    {
      :
    }
```

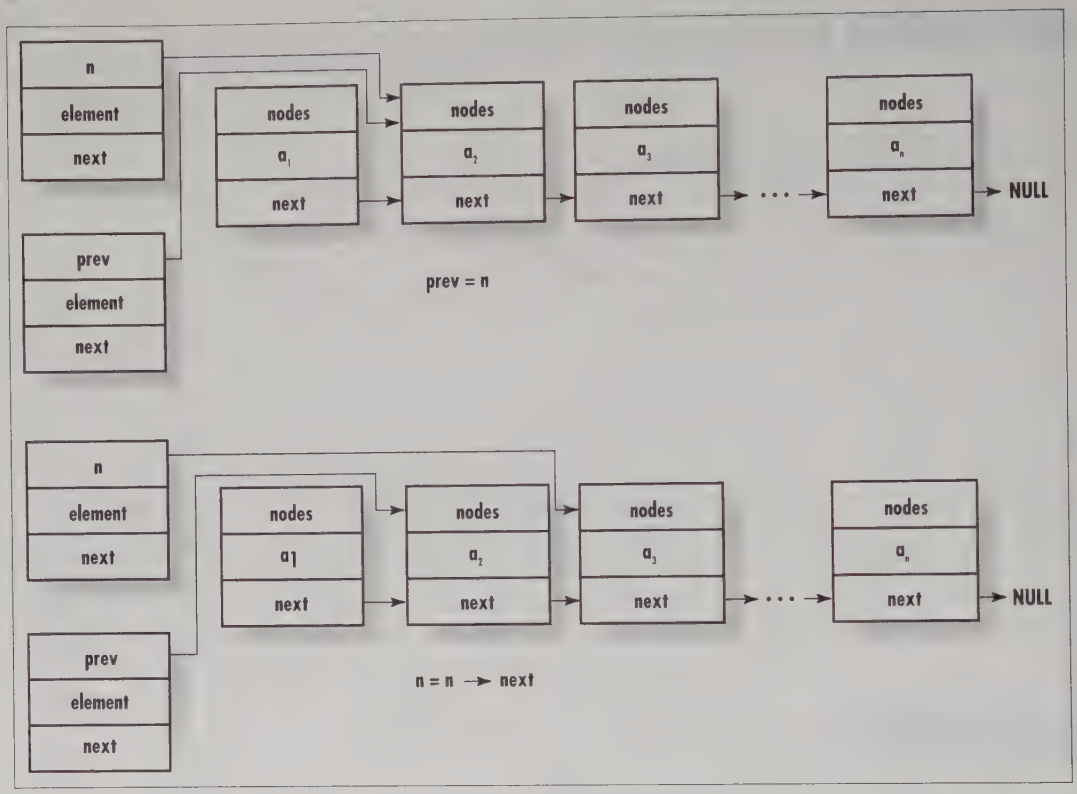


FIGURE 7-9 Advancing the node pointers while searching for an element instance

When an instance has been located in the list, you assign the *prev* node object's *next* pointer to the node object *n*'s *next* pointer. Since *prev* is the node object in the list just before the node object *n*, you are telling the *prev* node object to point one object beyond the node object *n*. Figure 7-10 shows the pointer assignments when you have located an instance of the element a_2 in the list. The node object pointer *n* is referencing the node to be removed. The node object pointer *prev* is referencing the node object before the node object pointer *n* is referencing. The node object *n* is pointing to the node you want to remove. So the *prev* node object pointer is going to keep one finger on the previous node and reach around with its next finger and point to the node just after node *n* in the list.

You can now use the C++ delete operator to discard the node object that *n* is referencing. The node object pointer is then assigned to reference the node that node object *prev*'s *next* pointer is referencing. This concludes removing an instance of an element in the middle of the list. You need to tackle the final case where the instance is located at the end of the list object.

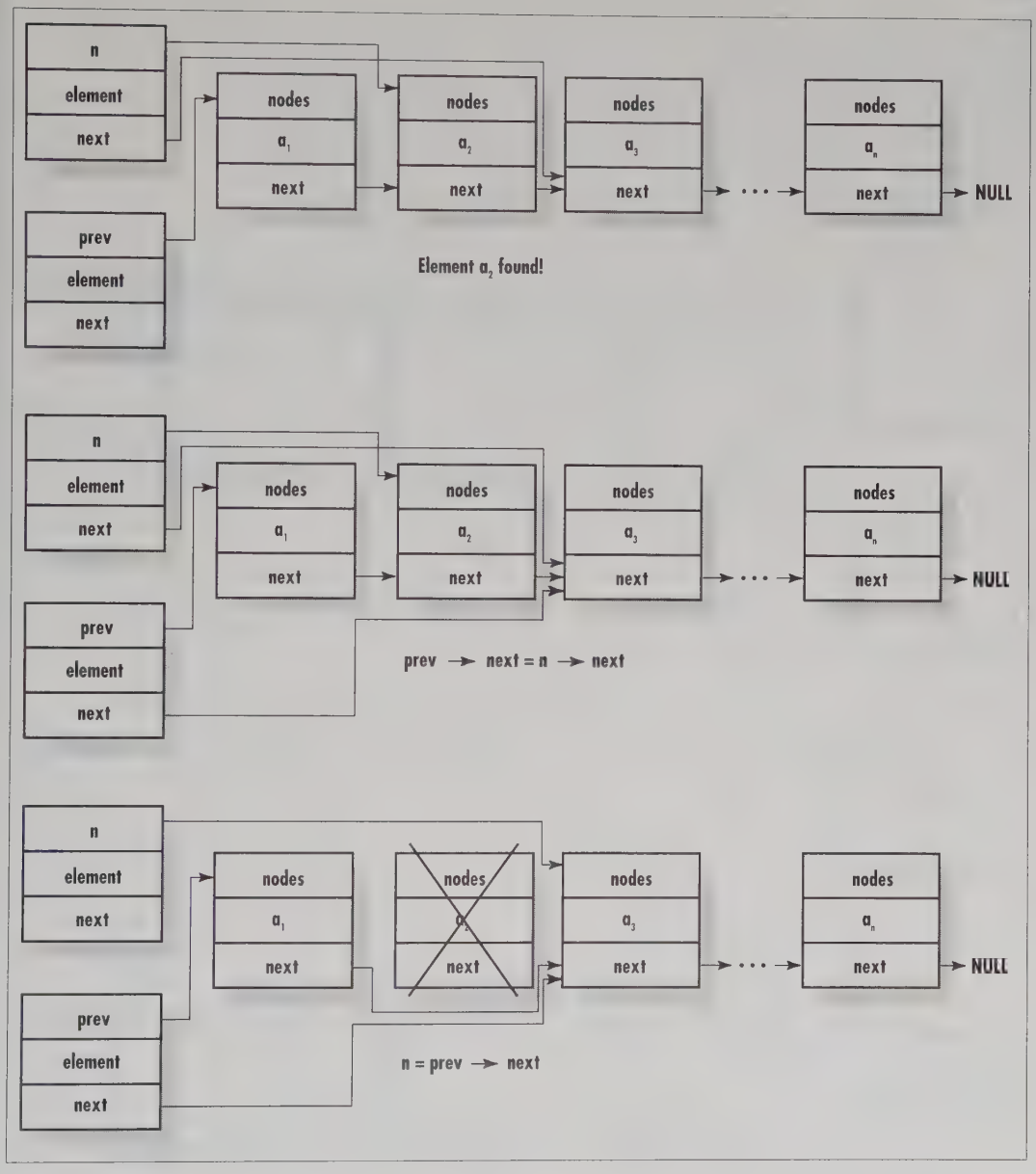


FIGURE 7-10 Removing the a_2 instance from the list

An instance is determined to be located at the end of a list if object pointer *n*, which is used for searching, is referencing the same pointer as the node object pointer *tail*. This situation is illustrated in Figure 7-11.

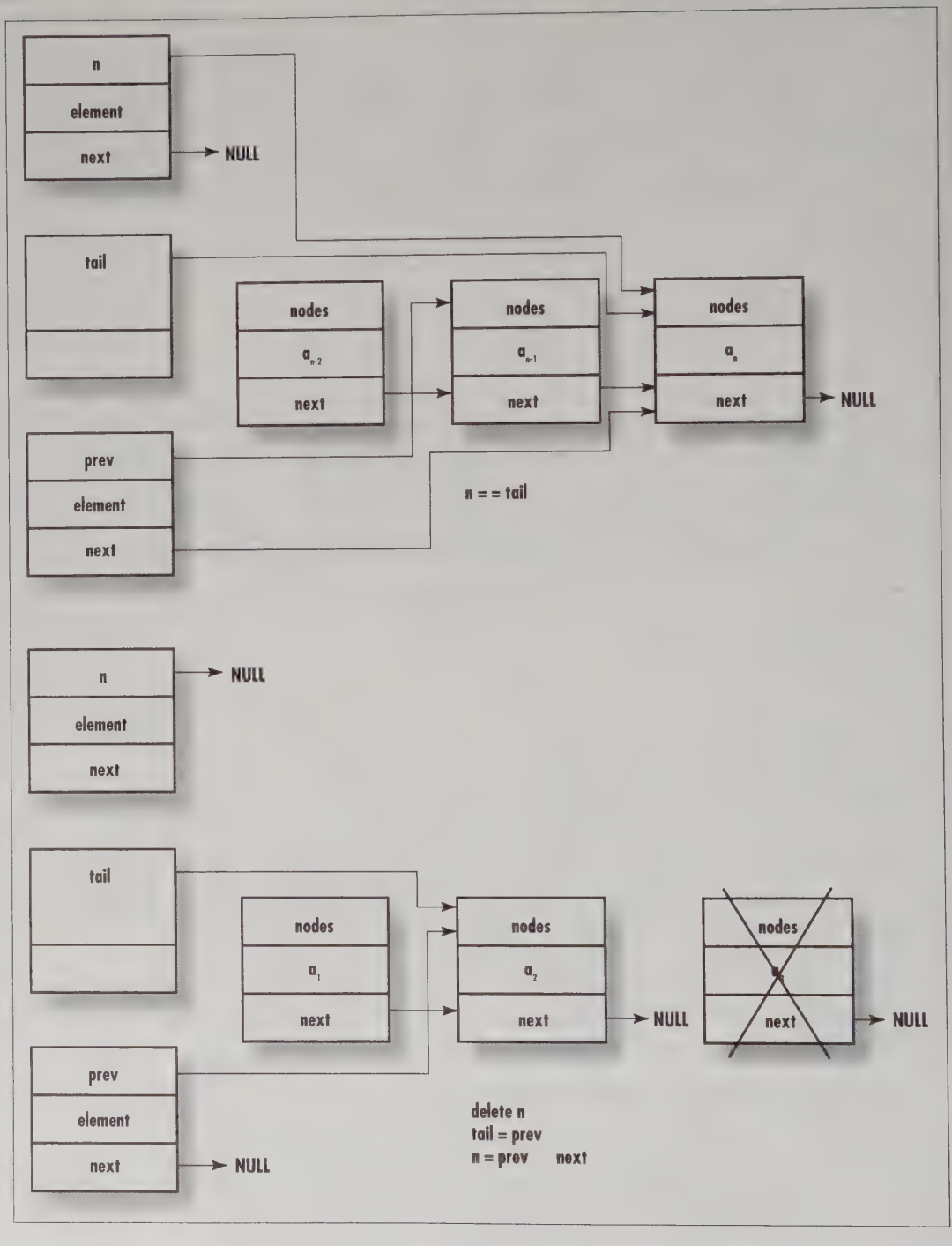


FIGURE 7-11 Removing an instance from the end of a list

The *prev* pointer places its finger on the next to last node in the list. The node object pointers *n* and *tail* are referencing the last element in the list. You wish to remove this instance of the element based on the results of your search criteria. So you tell the node pointer *tail* to back up and make the node object referenced by *prev* to be the last node in the list. The object referenced by *n* is then removed. The loop terminates when the pointer *n* equals NULL. Listing 7-13 shows how to handle all of these cases.

LISTING 7-13 Removing the item from the list

```
cListNode *prev = n;
n = n->next;

// trivial cases failed. search for the element
while( n != NULL )
{
  // element found?
  if (n->dataPtr == p)
  {
    prev->next = n->next;
    if (tail == n) // is element at the end of list?
      tail = prev;
    delete n;
    n = prev->next;
    elementFound++;
  }
  else // not found, go to next node
  {
    prev = n;
    n = n->next;
  }
}
return elementFound;
```



Modifying Academic Delusions into Something Useful

I know I'm in perfect company by stating the ADT lectures and assignments in school were pretty mundane. I mean, how many times in your professional career have you used a linked list to solve a polynomial? How about those database assignments where you read all those student records in from a file? Well, those days are long gone. It's time to learn real world applications for your ADTs. Before doing so, you need to modify your cList and cListNode classes to handle real world programming applications. This means you are weaning yourself off of the simple integer element data type and replacing it with a type void* pointer, as shown in Listing 7-14.

LISTING 7-14 Using the void* pointer to reference an arbitrary data type

```
class cListNode {
friend class cList;
private:
  void* dataPtr;   // pointer to data element
  cListNode *next; // pointer to next node object

  // default constructor
  cListNode( void* p, cListNode *node = NULL) { dataPtr = p; next = node; }
};
```

Likewise, your class declaration for cList needs alteration as well:

```
class cList {
private:
  cListNode *list;  // pointer to the head of the list
  cListNode *tail;  // pointer to the tail of the list

public:
  // default constructor
  cList() { list = NULL; tail = NULL; }
  // default destructor
  ~cList() { DeleteList(); }

  void InsertList( void *p ); // insert element into list object
  int RemoveList( void* p );  // remove element from list object
  void DisplayList();              // write the elements to DOS console
  void DeleteList();               // delete all the elements in the list
};
```

The classes are nearly identical to the original academic counterparts from earlier in this section. What's the difference? Well, my friends, you just crossed the fine line between an academic example and a serious ADT to handle a battalion of mechanized warriors. Listing 7-15 shows how you could create a mechanized robot battalion list object from the depths of your technologically deranged minds.

LISTING 7-15 Example of a list object for a mechanized robot battalion

```
// ppc Cannon
struct ppcWeapon {
  int rounds;
  int heat;
  int armorHealth;
  time_h delayOverHeat;
  int operational;
}
// shoulder mounted gattling gun
struct chainGuns {
```

```
  int rounds;
  int operational;
  int damage;
}

// jump jets
struct jumpJets {
  double vx;
  double vy;
  double mass;
  double speed;
  double friction;
  int damage;
}

// armor piloted warrior
struct mechRobot {
  int damage;
  int alive;
  int ammo;
  int armorHealth;
  jumpJets* thrusters;
  ppcWeapons* lasers;
  chainGuns* weaps;
};

cList l;

mechRobot *iBattalion[MAX_ROBOTS];

for (int ix = 0; ix < MAX_ROBOTS; ix++)
{
  l.InsertList(iBattalion[ix]);
}
```

This is purely an illustrative example of what can be done with lists in a video game. The beauty of these data structures is that they have nothing to do with boring student records and grades. Iterating through a list of robots, you could literally command a battalion of robots or warriors. Remember that cool game, *Space Hulk* from Electronic Arts, where you commanded the group of space marines into the gene stealer's hives? Guess how those warriors were being controlled? The sky is the limit on what can be done with the list object. I have provided a sample listing that adds and removes integers using your modified list object. In Chapter 8, you'll revisit your cList class to build a list of polygons for an Asteroids game. Some professional development environments may not use object-oriented programming; this is merely a design decision by upper management. If this is the case, the programming example in Chapter 9 uses the classical linked list. The discussions in this chapter for insertion, searching, and removal will still apply. The cList driver code is provided in Listing 7-16.

LISTING 7-16 Windows 95 Console window driver for the cList class

```
#include <stdlib.h>
#include "clist.hpp"

void main()
{
  cList l;
  int iArray[10];  // array of 10 integers

  for (int ix = 0; ix < 10; ix++)
  {
    iArray[ix] = ix;              // array[ix] = ix
    l.InsertList(&iArray[ix]); // insert element at array[ix]
  }
  l.RemoveList( &iArray[2] );  // arbitrarily remove element
  l.DisplayList();          // display the list object to console
}
```



Linked Lists in Review

Lists are a mathematical model representing a sequence of elements. A linked list is a computer data structure to implement the list model. A list allows flexibility in programming because it can grow and shrink dynamically. Operations performed on lists, such as insertion, searching, and removal are called abstract data types (ADTs). The list ADT can easily be implemented in C++ as a class.



Binary Search Trees

Another fundamental data structure in game programming is the binary search tree. Classically, binary trees have been used in computer science to describe the UNIX operating system file hierarchy, database systems, and compiler designs. Binary trees have been used extensively in video games for everything from generating chess probability move maps to dictionaries for text adventure games to 3D rendering and sorting of polygons, such as those used in *DOOM*. A tree is another mathematical model, like the list, but it is non-linear in representation. As the name suggests, the binary tree is in the shape of an inverted tree as shown in Figure 7-12. Like the list, a collection of elements in a binary tree are called nodes. The topmost node in the tree diagram is called the root node. Nodes are connected in binary tree diagrams by lines, called edges. Operations on the binary tree include insertion, removal, and searching.

A special binary tree model exists called a binary search tree (BST). A BST is represented by a linked data structure such as the one shown in Figure 7-12. A BST is represented by root pointer to the root node of the tree. The first

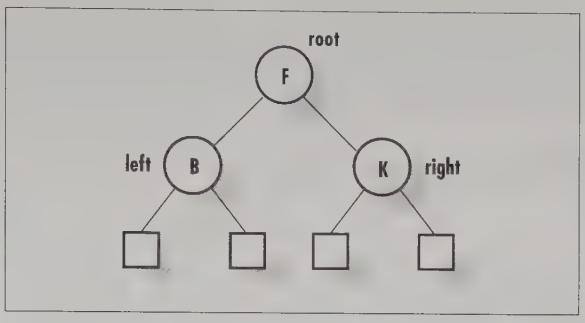


FIGURE 7-12
Simple binary search tree

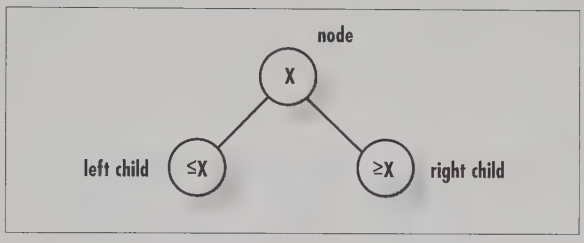


FIGURE 7-13
Building a BST

element inserted into the tree is the key element for the entire BST. In the case of the example in Figure 7-12, the character F was inserted into the BST first. Each element thereafter is inserted depending on whether the character is greater than or less than the character F. When the character K is inserted into the BST, logical comparisons determine that the character K is greater than F. Therefore, K will be inserted in the node to the right of the root, which holds the key element F. The insertion of the character B places this to the left of the root because B is less than F. You continue on in this manner, building the tree. Figure 7-13 shows the relationship of inserting nodes throughout the rest of the BST.



Object-Oriented Binary Search Trees

Whenever you pick up a good book on data structures, you'll find the names, Knuth, Ullman, or Hopcroft on the cover. In Jeff Ullman's text, *Foundations of Computer Science*, he uses ANSI C code to define a BST. The subtle changes you have made can be seen in Listing 7-17.

LISTING 7-17 BST structure using C code

```
typedef struct bstNode *bst;
struct bstNode {
  char element;
  bst right;
  bst left;
};
```

For your object, you want to retain the same design features, allowing your BST ADT to perform insertions, deletions, and searches. Listing 7-18 shows your tree node class design.

LISTING 7-18 Tree node class in C++

```
class cTreeNode {
  friend class cTree;
  protected:
    char element; // data type to hold in the tree
    cTreeNode *right;
    cTreeNode *left;

    cTreeNode( const char c = 0, cTreeNode *r = NULL, cTreeNode *l = NULL )
        :element(c), right(r), left(l) {};
};
```



In C++, a friend to a class, as in the *friend class cTree* declaration, gives the nonmember class cTree access to non-public members of class cTreeNode. This is a form of data hiding.

For your class design, you have a declaration for the class cTreeNode, which is the node object for the tree. Like its classical counterpart written in ANSI C, the node object has a right child pointer and a left child pointer. In this case, you have a right child node object pointer and a left child node object pointer. To build your tree object, by definition, you need to connect node objects. The methodical collection of these nodes builds your BST. In your class definition of a tree, you must also provide a root node object pointer from which to build the BST. Your example declaration of cTree class is shown in Listing 7-19.

LISTING 7-19 Tree class in C++

```
class cTree {
public:
  cTreeNode *root;

  cTree():root(NULL){};              // default constructor
  ~cTree() { root = NULL; };         // default destructor
  void MakeEmpty( cTreeNode* &n );
  void InsertTree( const char c, cTreeNode* &n );
  void InorderTreeWalk( cTreeNode* &n, unsigned int l );

};
```

The basic operations provided in this example are MakeEmpty(), which deletes the entire tree object, InsertTree(), and InorderTreeWalk(). The member function InsertTree() obviously adds elements to the tree and InorderTreeWalk() is a member function that walks the tree object and displays

the elements found on its walk. Before you proceed, there is one thing I must mention up front, and that is that all these functions, excluding the constructor and destructor, are recursive. I think somebody out there just did the Elvis lip. Recursion is ugly, but it's powerful.



Recursion

A recursive function is defined as one that calls itself from its own body of code. There are two kinds of recursion, direct and indirect. A direct function, for example foo(), has a call within itself to foo(). An indirect recursive function calls foo(), foo_1(), foo_2(), ... foo_n(), which then calls foo().

The greatest advantage to recursion is that once it's mastered, it can save a lot of code. The code recursion usually replaces its iterative code. An example would be a search routine using a while loop. Although not all iterative code is a candidate for recursion, there are some very particular cases. A drawback to recursion is that it can be time-consuming. An example of this is the indirect recursive function calls where foo() calls foo_1() and so on until the function foo_n() calls foo(). The code may go several layers deep before returning with a result that an iterative function might have found quicker. The up side to using recursion today is that compilers are optimizing function code generation and, secondly, the machines are getting faster.



BST Insertion

Inserting an element into the BST is probably one of the simplest examples of recursion. Begin your insertion by determining if a root node exists. If it does not, the value being inserted into the BST becomes the first element and its node is then the root of the tree. Books about data structures commonly refer to this element as the key. All other insertions hereafter are based on comparisons to the root key element. If an element is being inserted into an already existing tree, it is compared against the key element. This comparison determines whether the element will be inserted to the right node object or to the left object of the tree. As you add more branches to your tree, subtrees will perform this check to their parent node and so on. The code to recursively insert in a BST looks like Listing 7-20.

LISTING 7-20 cTree member function InsertTree()

```
void cTree::InsertTree( const char c, cTreeNode* &n )
{
  if (n==NULL)
  {
    n = new cTreeNode( c );
```

continued on next page

continued from previous page

```
    if (n == NULL)
    {
      printf("Error: <InsertTree( const char c )>\n");
      exit(1);
    }
  }
  else
    if (c < n->element)
      InsertTree(c, n->left);
  else
    if (c > n->element)
      InsertTree(c, n->right);
}
```

You can check to see if a tree is empty by examining the node pointer to the root object. Listing 7-21 shows the code you use to see if a tree is empty.

LISTING 7-21 Code used to check if the tree is empty

```
if (n==NULL)
{
  n = new cTreeNode( c );
  if (n == NULL)
  {
    printf("Error: <InsertTree( const char c )>\n");
    exit(1);
  }
}
```

The variable *n* is a pointer to the root of the tree object. When the pointer *n* is NULL, the tree is EMPTY. The cTreeNode constructor is called to initialize a node object. Figure 7-14 displays the results of inserting the element F into an EMPTY BST. Since this is the only element in the BST, the right and left branch node object pointers are NULL. This creates your tree and each insertion thereafter will be based on the contents of your root node object.

Now proceed with inserting the element Q into a non-empty BST. The first item you inserted was the element F. The element Q is compared against the root node object element as shown in Listing 7-22.

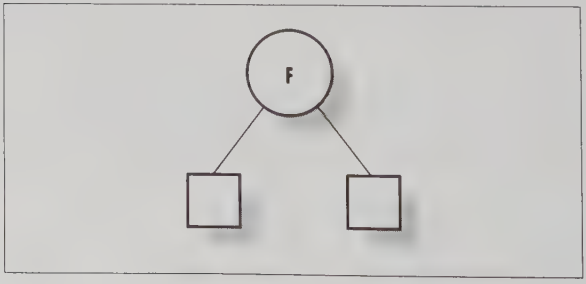


FIGURE 7-14
Inserting the key into the BST

LISTING 7-22 Inserting an element into the tree

```
if (n==NULL)
{
  // if tree object EMPTY
}
else
if (c < n->element)
  InsertTree(c, n->left);
else
if (c > n->element)
  InsertTree(c, n->right);
```

Obviously Q is greater than F so the insertion will be performed on the right side of the tree. But, from your handy Figure 7-14, you see that the right child object pointer is NULL when you recursively call InsertTree(). The function treats this in the same manner as it did when you inserted F. The cTreeNode constructor is called and initializes a node object for Q. The tree node is referenced as the right sibling of F. Did you ever wonder about BST sibling rivalry? While you ponder this dilemma, take a look at Figure 7-15, which shows the insertion of Q into the BST object.

When you insert the element M into the tree, examine the element in the root node. The element M is greater than F so you recursively call InsertTree (M, n->right). You are now at the node object which contains Q. The element M is less than Q so you recursively call InsertTree(M, n->left). The left subtree of node object Q is NULL. The constructor is called to initialize a new node object and its right and left node object pointers are initialized to NULL. The resulting tree object is displayed in Figure 7-16.



Inorder Tree Walking

A property of the BST is to print out the elements of the tree in sorted order. Of course this involves recursion, but not to worry. Remember, you inserted the elements into the tree object in a special order. The element in the left subtree of its parent is less than the parent's element. Likewise, the element in

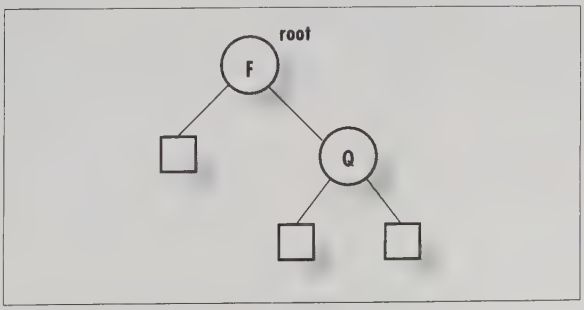


FIGURE 7-15
Inserting Q into the BST

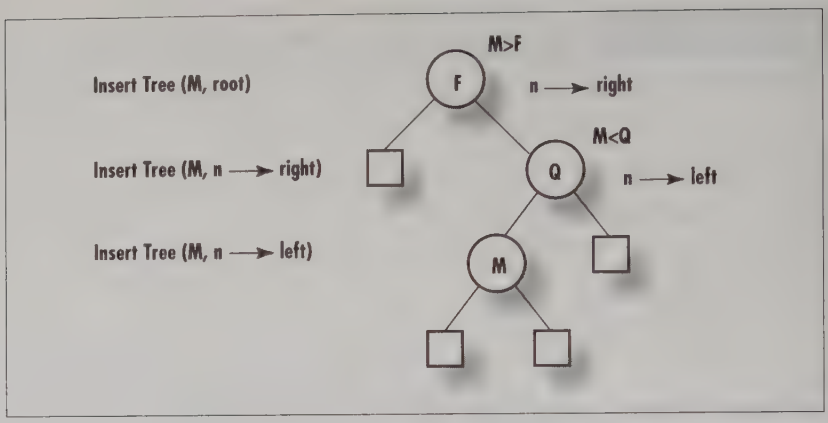


FIGURE 7-16
Inserting the M element into the tree

the right subtree is greater than its parent's element. In Figure 7-17 the root contains the element F, which is the root node. All the nodes in the subtrees to the left of the root node contain the elements that are less than F. These are the elements A and B. In the right subtrees, the elements are greater than F and these elements are M, Q, and Z. The recursive Inorder Traversal algorithm is provided in Listing 7-23.

LISTING 7-23 Recursive Inorder Traversal of a tree

```
if (n != NULL)
{
  InorderTraversal(n->left)
  print element
  InorderTraversal(n->right)
  print element
}
```

The algorithm continues down the left edges of the tree until a NULL n->left is reached. In the example, the element A is in the leftmost subtree. The A will be printed first. Since you are layers deep in recursive function calls, return to the next recursive caller. In this case you are at the node containing the element B. The element B is printed. The rightmost subtree of B is empty. Return the recursion stack, if you will, to the next caller. You are at the root and you print the element F. On the right-hand side of the tree you again proceed down to the leftmost subtree. You discover the element M there and you print it. Backing up the recursion stack caller frame, you arrive at the element Q and

print this. Finally, you process a right subtree and print the last element in the tree, the element Z. This is a bit confusing, so go ahead, check out the recursion road map in Figure 7-17. The listing for InorderTreeWalk() is shown in Listing 7-24.

LISTING 7-24 cTree member function InordertreeWalk()

```
void cTree::InorderTreeWalk( cTreeNode* &n, unsigned int l )
{
  unsigned int i;

  if (n == NULL)
    return;
  InorderTreeWalk(n->left, l+1 );
  for (i=0;i<l;i++)
    printf("  ");
  printf("%c\n",n->element);
  InorderTreeWalk(n->right, l+1 );
}
```

The source code driver for a Windows 95 console application is presented in Listing 7-25.

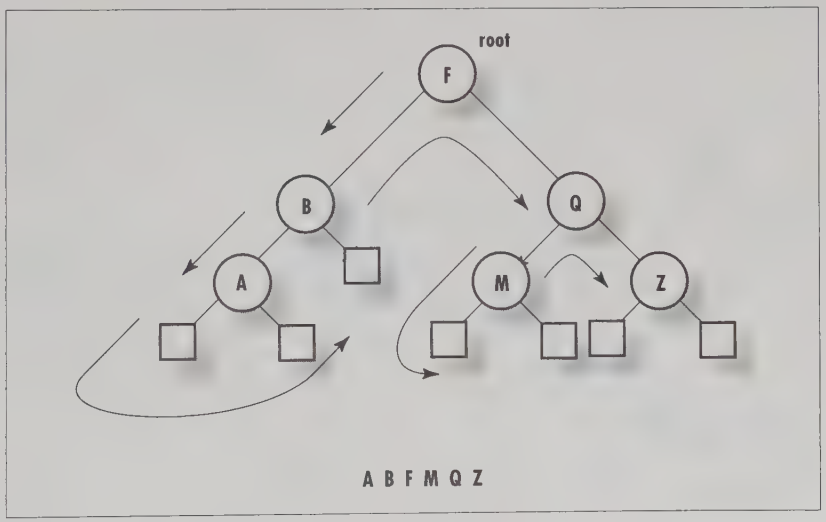


FIGURE 7-17
Inorder tree walking

LISTING 7-25 Windows 95 console window driver for the cTree class

```
#include <stdio.h>
#include "ctree.hpp"

////////////////////////////////////
//  global variables
////////////////////////////////////
char *array = "dbacfeg";

////////////////////////////////////
// main code body
////////////////////////////////////
void main(void)
{
  cTree bt;  // class cTree object declaration
  char c;     // element to insert

  // walk the array and insert into tree
  do
  {
    c = *array;
       bt.InsertTree(c, bt.root);
       array++;
  }while(*array);

  bt.InorderTreeWalk(bt.root,NULL);
}
```



Binary Search Trees in Review

Binary trees are a model to represent hierarchical information such as directory structures and dictionaries. A binary tree has a left child and a right child. A binary search tree is a special tree where nodes stored to the left precede the node key and nodes stored to the right follow the node key. Recursive functions lend themselves very well to tree data structures.



Game Program Flow

In this section, you'll look at game program flow, tasks, and threads. Program flow is critical in game design because you must control how the program executes. Windows 95 game applications redefine how DOS and Windows 3.1 programmers are going to design their programs. The Windows 95 operating system is a multitasking operating system and can run one or more programs concurrently. The topics in this section will focus on how to use multitasking in your Microsoft Direct X Game SDK applications.

Our game application is a *process* consisting of code and data. The Windows 95 operating system allocates system resources to each process. When the user double-clicks the icon for the game application from the Windows 95 explorer, the operating system creates a process and a thread. The process is the instance of the application and the thread is the path of execution of the application. The operating system instructs the CPU to start executing the thread with the start-up code, which in turn calls WinMain(). CPU time is scheduled using threads.



Windows 95 Multitasking

The Windows 95 multitasking environment is different from its Windows 3.1 predecessor. The common form of multitasking is for the operating system to yield a time slice of the processor for each task running. Instead, Windows 3.1 uses messages to swap between applications. Under Windows 3.1, applications are scheduled using *nonpreemptive* multitasking. Applications work cooperatively under the Windows 3.1 operating system by yielding the processor with a call to GetMessage(), PeekMessage(), and WaitMessage().



Nonpreemptive Multitasking

In the Windows environment, most applications have a central GetMessage() loop that allows the application to cooperate with the other applications running in the system. The message loop you wrote for your Windows 95 generic game is how you use nonpreemptive multitasking in your game application. The call to GetMessage() from your message loop will return a message if one exists in the application's queue. Normal Windows applications use the GetMessage() call exclusively in the *while* statement. This implementation allows the application to yield the processor to another application if no message is waiting in the queue to be processed. For game applications, your generic game application uses PeekMessage() to see if a message is waiting. If there isn't a message, your application does not yield the processor to other applications. In a moment I will discuss how to yield the processor when a game application is running in an exclusive mode.



Preemptive Multitasking

The Windows 95 operating system is a preemptive multitasking environment. In a preemptive multitasking environment, the processing time is divided into slices. An application running on Windows 95 is called a thread. When a thread has used up its slice of processor time, it is suspended, allowing another thread to use the processor. This process involves saving the context of the thread being

suspended to a queue and resuming the context of the thread to be executed. Each thread has a processor time slice of approximately 20 milliseconds or until it blocks on I/O or waiting for a system object. Giving each thread a finite time in the processor is how Windows 95 achieves its multitasking. The operating system can become overburdened with threads if too many are executing.



Scheduling Tasks and Game Events

Designing a Windows 95 game application allows you to take advantage of both preemptive and nonpreemptive multitasking features. Just as the operating system schedules tasks and events for processor and system resources, you can control event and program flow by scheduling from within your application. The type of tasks and events you want to schedule are screen updates, splash screens, game pausing, game application termination, executing threads, dlls, and minimizing and maximizing the game application to relinquish the processor to another application. Using this design scheme, your new message loop is shown in Listing 7-26.

LISTING 7-26 Windows friendly message loop

```
while(1)
{
  if(!bIsActive || PeekMessage(&msg, NULL,0,0,PM_NOREMOVE))
  {
    if(!GetMessage (&msg, NULL, 0, 0))
      return msg.wParam;
    TranslateMessage (&msg);
    DispatchMessage (&msg);
  }
  else
  {
    // do critical game stuff
    if (bIsActive)
      scheduler();
  }
}
```

The game application message loop is nearly identical to the loop introduced in Chapter 1. The key differences are the addition of the Boolean variable *bIsActive*, which you'll use to determine if the application is minimized or maximized, and also the function scheduler.



Designing a Game Process Scheduler

The scheduler executes an event by examining a global variable called *taskState*. The *taskState* contains one of any values defined in an enumerated list of task states. When the application initializes, the taskState is set to TS_SPLASH. When the scheduler is called, it reads the *taskState* and executes the code for the splash screen. When the user presses the enter key from within a *taskState* of TS_SPLASH, the *taskState* is set to TS_CRITICAL to begin executing the game events and tasks. The code for a simple scheduler is shown in Listing 7-27.

LISTING 7-27 Game process scheduler

```
enum {TS_CRITICAL, TS_SPLASH, TS_SUSPEND, TS_TERMINATE};
int taskState = TS_SPLASH;
void scheduler()
{
  switch(taskState)
  {
    case TS_SPLASH:
      Splash( "Splash.bmp");
      break;
    case TS_CRITICAL:
      updateFrame();
      break;
    case TS_SUSPEND:
      break;
    case TS_TERMINATE:
      PostQuitMessage(0);
      break;
    default:
      break;
  } // switch taskState
}
```

The values in the enumerated list can be added or removed as your design requires. This is just a simple list provided to demonstrate the workings of the scheduler function. Now take a look at how to use these states in your game application.



State TS_SPLASH

In the example code, you set the *taskState* variable to TS_SPLASH in the game application initialization code. When the scheduler is called, it will display the splash screen. This is the body of code you may want to use to launch your game

intro screens and menus. The example calls the *Spells of Fury* DDTOOLS function Splash to display a game intro screen, as shown in Figure 7-18. The *taskState* does not change until the user presses the ENTER key to change the *taskState* variable to TS_CRITICAL. The code to execute the state transition is in the WinFunc() code. Here you check the *wParam* field of the message structure for the value VK_ENTER on a WM_KEYDOWN message. Listing 7-28 shows how you handle the event when VK_RETURN is pressed by the user and the task state is TS_SPLASH.

LISTING 7-28 Task state TS_SPLASH

```
case VK_RETURN:
   switch(taskState)
   {
     case TS_SPLASH:
       // set the palette from the BMP
       lpDDSPrimary->SetPalette(lpDDPal);
       lpDD->FlipToGDISurface();
       taskState = TS_CRITICAL;
       break;
     default:
       break;
   }
break;
```

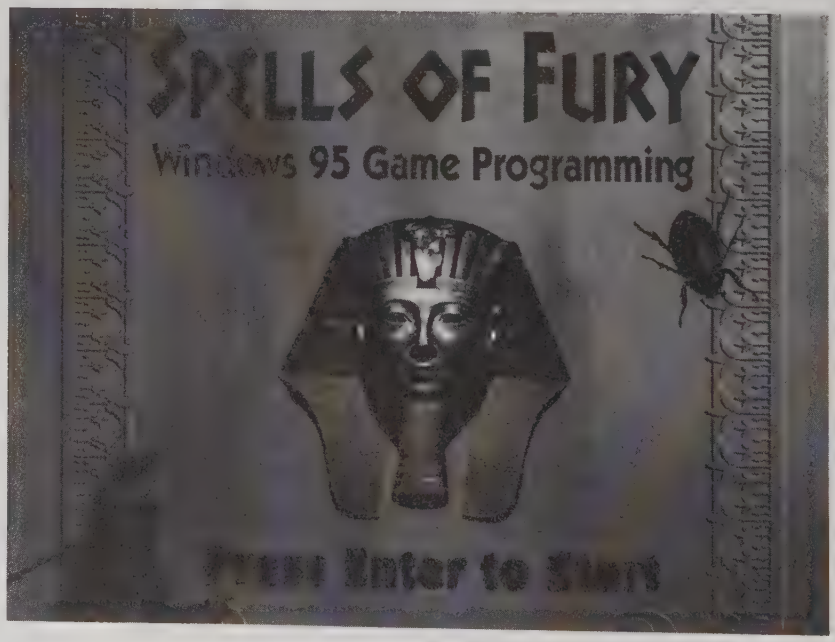


FIGURE 7-18
Spells of Fury splash screen

If the *taskState* is set to TS_SPLASH when the enter key is pressed, the *taskState* is changed to TS_CRITICAL. Note that in this state you instruct DirectDraw to flip to GDI video memory. The splash screen is static and there is no sprite animation so you have no screen updating to process. You may also wish to use Windows dialogs for user configuration and setup as well.



TS_CRITICAL

The *taskState* set to TS_CRITICAL is where you process your game application events. These include sprite animation, network communication, Direct X library calls, and user input handling. When your game application is not processing Windows messages, it is most likely in the executing critical game code. A simple example of the critical game code is shown in Listing 7-29.

LISTING 7-29 Critical game code in the updateFrame() function

```
void updateFrame()
{
  checkKeyboard();
  EraseBackSurface();
  drawTerrain();
  drawBubonic();
  flipSurface();
}
```

The scheduler calls the function updateFrame(), which is actually the code used back in Chapter 4 to demonstrate keyboard input. The functions you call from updateFrame() include checkKeyboard(), drawTerrain(), drawBubonic(), and the call to flipSurface(). As you can see, the critical task section of your code handles the user input and game rendering.



Minimizing a Game Application

The ideal operating conditions for a Microsoft Game SDK application is executing in an exclusive mode. Just because your game application is utilizing most of the system resources doesn't mean that the user may not want to minimize the game application, execute another Windows 95 application, and then come back to your game. As a developer, you are responsible for providing a *multitasking-friendly* and *well-behaved* game application.

A standard Windows 95 application minimizes when the user double-clicks the mouse cursor on the minimize box on the application's title bar. Your game applications don't have a title bar or, for that matter, a minimize box. This example will provide this functionality by allowing the user to press the F12

function key on the keyboard to minimize the game application. You need to provide code in your generic game application's WinFunc() message handling to check the WPARAM field for the value VK_F12 on a WM_KEYDOWN message. This code looks like this:

```
case VK_F12:
       ShowWindow(hwnd, SW_MINIMIZE);
break;
```

You used the Windows API call ShowWindow() to minimize your application. When this code executes, the game application is minimized to the system menu bar. Processor resources are relinquished for another Windows 95 application to execute. As you shall see momentarily, just because the application is minimized doesn't mean it stops executing.



Maximizing the Game Application

When I first started writing code to minimize and maximize game applications, I ran into two problems which were very irritatingly related. When I minimized the game application, the code for the game would sometimes continue executing in the background. So I put in a few flags to pause the game when I wanted to minimize the game application. When I went to maximize the application, I couldn't get the pause flags to toggle properly and resume the game. I spent a night hacking away at clever schemes to get around this problem. It turned out I was thinking too hard. All I really wanted to do was to pause the game events when the game application wasn't active. I only needed to check for a message that Windows 95 would send when it maximized the game application. The Windows message that is sent by the operating system whenever an application is activated is WM_ACTIVATEAPP. When I checked with Windows game developers on the net, they insisted that the Windows API call IsWindowVisible() would solve my problems. This did, but not reliably. Good ole' WM_ACTIVATE APP works like a charm. Remember the Boolean flag, *bIsActive*, introduced earlier? Do you want to take a clever guess where to set this? The following code, called from WinFunc(), should remove any cloud of mystery on this matter:

```
switch (message)
{
  case WM_ACTIVATEAPP:
    bIsActive = (BOOL)wParam;
     break;
  :
}
return DefWindowProc (hwnd, message, wParam, lParam);
```

The WPARAM field you examine is the WM_ACTIVATEAPP activation flag. When a Windows application is minimized, this field is set to 0. A non-zero value indicates the window is active. Using the LPARAM field of the

message, you can also retrieve the thread handle to this application. I didn't need this detail of information; I just wanted to suspend my game events scheduler until the game application was active (in this case, visible).



TS_SUSPEND...Pausing a Game

Most games have a pause button. In this case, you want to pause the game but leave the game application in exclusive mode. To do this, you will set the *taskState* to TS_SUSPEND, where the scheduler doesn't process any input or game events. The most natural key to pause a game with is the P key. I have provided a key definition in header file gamekeys.h to utilize the A through Z keys in your game application. The code to trigger a game pause of P key down is shown in Listing 7-30.

LISTING 7-30 Pausing a Direct X game

```
case GAME_KEY_P:
  switch(taskState)
  {
    case TS_CRITICAL:
      taskState = TS_SUSPEND;
      break;
    case TS_SUSPEND:
      taskState = TS_CRITICAL;
      break;
    case TS_SPLASH:
    default:
      break;
}
break;
```

All the P key does in your code is toggle the *taskState* from TS_CRITICAL to TS_SUSPEND and vice versa. This is how most game applications handle a paused game. Press the P key to pause the game and then press the P key again to resume the game. Take a look at *Heretic*, *DOOM*, and *Raptor*, to name a few PC games that use this game pausing convention.

When the taskState is set to TS_SUSPEND, the scheduler doesn't execute any code. Here you may want to call a special menu for when your game is paused, or blit the string 'paused' to the display.



TS_TERMINATE

The game application terminates gracefully with the taskState set to TS_TERMINATE. All the applications in *Spells of Fury* terminate when the Esc key is pressed. This results in immediate shutdown of the game application. I kind of like the idea of allowing the game application to terminate on demand.

Every now and then, we software engineers like to conduct our own marketing survey of the competition's game software and when our manager sneaks up behind us during one of our unofficial evals, its kind of nice to get out of the application as quickly as possible. For this reason, I like the rapid game-termination key which few game developers seem to utilize.

Your code checks for the virtual key, VK_ESC, to be pressed by the user. The code for your WinFunc() message handling looks like this:

```
case VK_ESCAPE:
  taskState = TS_TERMINATE;
  break;
```

When the taskState is set to TS_TERMINATE, the scheduler executes the following code:

```
void scheduler()
{
  switch(taskState)
  {
          :
    case TS_TERMINATE:
      PostQuitMessage(0);
       break;
    default:
      break;
  } // switch taskState
}
```

The scheduler calls the Windows API routine PostQuitMessage(), which posts a WM_QUIT message in the game application's message queue. This call signals Windows 95 that your application is requesting to terminate. The parameter of 0 is your game application's exit code.

The scheduler is just one of many techniques that can be used by a Windows 95 game application when it's executing in exclusive mode. The simplicity of the function itself makes it versatile. Each event in program execution can be labeled as a *taskState*. New states can easily be added to the scheduler as needs arise. This can include adding new threads or maybe a DLL or two to the scheduler. Go ahead and experiment with the code.



Scheduling Tasks and Game Events in Review

The Windows 95 operating system uses preemptive and nonpreemptive multitasking. Windows 3.1 uses nonpreemptive multitasking, where applications yield the processor by sending messages to the operating system. In the Windows 95 environment, 32-bit applications use preemptive multitasking, where each application is assigned at least one thread. Each thread is given a finite time slice in the processor.

A game application can yield its exclusive mode by forcing its window to minimize. The application can become active on a window MAXIMIZE by

handling the Windows message WM_ACTIVATEAPP. A game application can terminate by posting a WM_QUIT message using the Windows API call PostQuitMessage().



Concurrent Threads

Now comes a completely new concept to PC game design, multithreaded game applications. Using threads in your game design, you can manage how your game executes. For instance, say your game uses an id Software-style WAD file that is very popular these days. If your game is CD-ROM–based, there is significant lag time when the WAD is being loaded in. You could minimize the lag time in a Windows 95 game by creating a thread to go out and load the WAD while your game players are still fighting to get to the portal to teleport them to the next level. What you have is two or more threads from the same game executing concurrently. If the lag time between levels is minimized, you are assured that the players' adrenaline remains elevated. In this section, you will examine using threads to initialize your DirectInput device while your main application thread loads the splash screen.

To create the thread, you will make some modifications to your scheduler function. You need to add the state, TS_NEW_THREAD to your enumerated list of task states. Also, it would make sense to add a new enumerated list of thread types you will be creating. The thread list contains the type of threads you will activate from your game application. These include reading DIBs, initializing DirectX devices, and reading WAD files, to name a few. You have the following:

```
enum {TS_CRITICAL, TS_SPLASH, TS_SUSPEND,
                   TS_TERMINATE, TS_NEW_THREAD};
enum enumThreadList { TL_NONE, TL_INIT_JOY, TL_READ_WAD_FILE };
```

The body of code you need to add to your scheduler function is shown in Listing 7-31.

LISTING 7-31 Launching a thread using the scheduler

```
case TS_NEW_THREAD:
  switch(threadActivate)
  {
    case TL_INIT_JOY:
      CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)InitJoyThread,
                                  (LPVOID)NULL,0, &threadID);
      break;
    case TL_READ_WAD_FILE:
    default:
      break;
  }
  break;
```

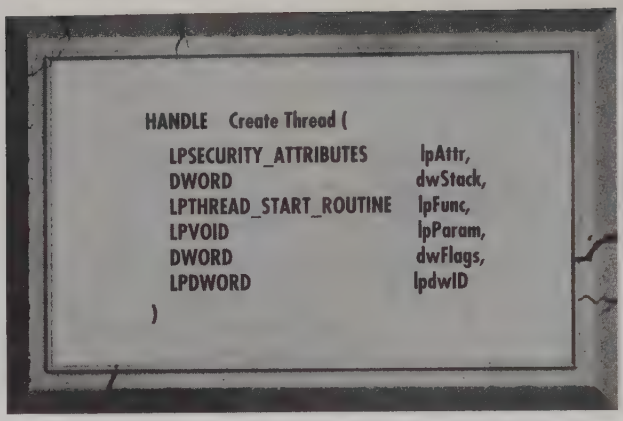


FIGURE 7-19
Windows 95 API CreateThread function

Inside the *case* statement to handle the taskState TS_NEW_THREAD, you have an additional *case* statement to handle the type of thread you are activating. The Windows 95 API call CreateThread() (see Figure 7-19) is used to activate your thread. To use the CreateThread() function, pass a NULL pointer to the security attributes argument. The thread function is operating concurrently with the primary thread, so the thread function has its own stack in the system. Remember, both the primary and child thread are vying for processor time. The CPU regards the threads as individual processes. So each thread will need its own stack. The CreateThread() function has a calling argument to set aside the stack size. You have no idea as to how big a thread stack should be, so you'll pass a value of 0. Windows 95 will give you the stack size of the primary thread. Since you have a stack, you must also provide information about the entry point of your thread. Remember from Chapter 1, the Windows 95 operating system needed information about the entry point of your WinMain() function. Tell the operating system to go look up the address entry point of your child thread using (LPTHREAD_START_ROUTINE) InitJoyThread(). The calling arguments passed to your thread are passed in through the parameter to CreateThread, LPVOID *lParam*. The *dwFlags* parameter allows you to control how your thread is to be executed. You could pass CREATE_SUSPENDED, to create and then suspend the process. A 0 value passed into this field executes your thread immediately. The last field is the thread ID. This is the unique ID for the operating system to determine what processing and what resources belong to which thread.



InitJoyThread()

When you use threads, you must use the thread function prototype, as shown in Figure 7-20. When you call CreateThread(), the thread of execution begins

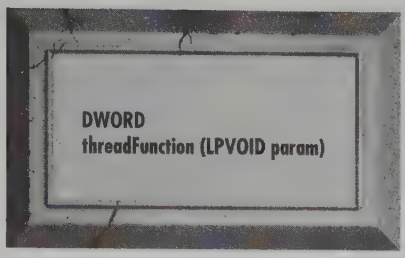


FIGURE 7-20
Windows 95 thread function prototype

with a call to your thread function. Your thread function is InitJoyThread. Its execution lasts for as long as it takes to execute your InitJoystick() code you used back in Chapter 4. The code for initJoyThread() is shown in Listing 7-32.

LISTING 7-32 Thread InitJoyThread()

```
DWORD InitJoyThread(LPVOID param)
{
  taskState = TS_SPLASH;
  initJoyStick();
  return 0;
}
```

In this example of a thread function, you immediately set the *taskState* to TS_SPLASH. This keeps the scheduler busy with calling the splash screen code. Note that this is not a process spawn; the splash screen initialization is vying for system resources at the same time the DirectInput initialization is executing. When the initialization is completed, the thread returns.

If any C library functions are being used, it is recommended that you use beginThreadNT(). Another caution to be aware of when using threads is the danger of corrupting shared resources or simultaneous access of shared resources from multiple threads. Complete discussions on multitasking and concurrent programming can be found in two good texts, *Concurrent Programming Principles and Practices* by George R. Andrews, *and Principles of Concurrent and Distributed Programming* by M. Ben-Ari.



Concurrent Threads in Review

Threads can greatly improve how you manage your game applications. Multithreaded multitasking allows for multiple threads to execute concurrently on one CPU. Excessive use of threads can exhaust or strain system resources. A thread is activated by a call to the Windows 95 API function, CreateThread. CreateThread calls a special thread function which the programmer provides. Threads can easily be activated from your scheduler game application function.



Conclusion

This was a pretty hefty chapter to read in one sitting, but you were introduced to, or refreshed with, some very important concepts. Linked lists are very common to games. A carpenter uses a hammer about as frequently as a game developer uses dynamic data structures. How you manage memory is equally important, especially on a multitasking operating system, where, in exclusive mode, you must anticipate your task may be swapped out. Whether it is because the manager is sneaking around or because you need to send a resume off to id Software using your e-mail, both Windows 3.1 and MS-DOS developers must become conscious of users who expect a commercial game application to be well-behaved in the Windows 95 environment. In the next chapter, you'll utilize a few concepts from data structures and your new scheduler function to build a simple *Asteroids* game.



UNLEASHING THE SPELLS OF FURY

8

ASTEROIDS



ASTEROIDS



The coin-op game that changed my life forever was *Asteroids* by Atari. There was something about that game with its annoying little heartbeat sound effects for the tumbling debris, and the pulsating whirs of the enemy saucers. I don't know how many quarters I plugged into it when I was a kid. The game is a classic. An art history book would cover the works of Leonardo da Vinci, Shakespeare, and Mozart; thus, it is only fitting that a book on video games should cover *Asteroids*.

Despite its vector graphics simplicity, *Asteroids* is renewing its appeal with a second generation of video game enthusiasts. *Asteroids* is also popular on UNIX platforms such as the Sun and SGI. In this chapter, you'll examine how to develop your own *Asteroids* game using the list objects you designed in Chapter 7.

EPISODE THE ZERO VIRUS



"13, did you notice that this corridor is altering to a gray-scale palette? Did you touch something?" I asked with great concern.

"Bary, pan over to the left. The corridor resolution is losing its integrity. This looks like the virus that was affecting the port in which you entered the artifact," 13 commented.

"Dr. Goldmen is nearby. She must be in the core memory chamber. Let's pick up the pace." I was eager to meet up with Carol. The two of us working together could definitely resolve this crisis.

"I don't recommend you attempt any contact with Goldmen. The virus is running its course. Bary, whatever she was is gone. She is now a destructive cognitive program executing until it wipes out everything in its path. I suggest you find a way out of here and head for the central processing center," 13 advised.

"You may be right. Look for a data bus. We're near the core memory. There's got to be some cache and a bus for Agamemnon to run his operations," I suggested. After I spoke, the door to the core memory began to deresolute. A gaping hole opened and I could see the core.

A wind with the power of a tornado spun violently inside. From the center, forks of lightning emanated, vaporizing the virtual guardians of Agamemnon. In the midst of the devastation levitated the form of a woman, the arms of the figure reaching straight out from her sides. Her body from the neck down was black as if she were an empty void. I recognized the face of the creature. It was Carol. Her hair stood on end from the electrostatic charge. It reminded me of a picture of Tesla and his electrostatic generator, but this scene was far more violent.

She had no eyes, just a radiance of white energy emanating from the sockets. 13 was right; my friend Carol was no more. Agamemnon teleported in armies of virtual monsters. I stood and watched as she destroyed even the most powerful of Agamemnon's abominations.

The winds were creating a vacuum and it started to suck me in. The howling was so loud I couldn't hear 13's instructions. I started to panic as I was drawn into the Zero Virus' cleansing wake.

I tried desparately to grab onto anything and anchor myself. But nothing worked. I was being pulled in. The empty, radiant eyes of Carol, now the Zero Virus, just glowed in the ecstasy of her godlike state.

In desperation, I turned to my own imagination for a resolution. I summoned, once more, the powers of Captain Cosmic. My muscles bulged to sizes greater than any of those found on *American Gladiators*. My epidermis was replaced with the cosmic super alloy armor. Summoning the powers of Captain Cosmic, my strength increased a hundredfold. I stood my ground in the fiercest of winds the Zero Virus could throw at me.

She too grew stronger in a impulse to take me on. I countermeasured my strength again to match hers. If the Grinch were watching for any power level spikes, I think I was now drawing enough energy to brown out an entire state.

I stepped into the core chamber to try to reason with Zero, to see if any essence of Carol was left. I did not know how long I could hold my new form, but I had to try to save my friend.

"Carol? It's me, Bary. Do you remember me?" I asked.

For a moment the radiant energy in Zero's eye sockets subsided and I saw Carol's eyes.

"Bary, you must leave now! Leave me. Leave here. Get out of the artifact, now. Agamemnon and I are now engaged in battle. He is losing ground. I plan to destroy him and the artifact! Now go. I am Carol no longer."

"Listen to me! Your old employer has different motives to keep Agamemnon online. I believe you and I have been successful because Agamemnon has found a new residence out on the Internet." My argument was cut short as Carol's eyes faded and the lifeless energy radiated in her sockets.

Zero unleashed her electrostatic fury on me and blasted me to the other side of the memory core. The chamber wall was incapable of stopping my virtual massive form and I went through the wall. Zero sealed the hole as I was flung into the adjoining hall.

My Captain Cosmic armor morphed back into my normal virtual self. I could hear 13 panicking for me to respond.

"Bary! Bary! Do you copy?" he pleaded with great concern.

"I'm here, 13. Just took a little detour. But I ended up closer to where I wanted to be."



Creating a Shape List

In the last chapter you constructed a simple linked list object for storing a list of integers. In this chapter you're going to modify your cList and cListNode classes to build a linked list of shape objects. You'll use the Shape class from the Chapter 6 game example *SpaceFoil*. Using the foundations of Chapter 6 and Chapter 7, polygon Shape objects and linked lists, you'll construct a vector graphics style *Asteroids* game.



The Shape List Object

The linked list lends itself very well to maintain a set of disordered objects such as tumbling space debris. However, the previous example of a list object is going to need some modification to allow you to reference Shape objects. The Shape objects that you used in the *SpaceFoil* game in Chapter 6 are the Shape objects you're going to use to define space debris (the asteroids), missiles, enemy ships, and of course your already-defined SpaceFoil fighter. From Chapter 7 you declared your linked list class using the code in Listing 8-1.

LISTING 8-1 The cList class from Chapter 7

```
class cList {
private:
  cListNode *list;

public:
  cList() { list = NULL; }
  ~cList();
```

continued on next page

continued from previous page

```
  void InsertList( int i );
  void DeleteList();
};

class cList {
private:
  cListNode *list;
  cListNode *tail;

public:
  cList() { list = NULL; tail = NULL; }
  ~cList();
    :
};
```

The list object, as listed, is providing you with simple operations. (See ADTs in Chapter 7, to insert and remove items from a list.) To maintain a list of Shape objects, you're going to modify your cList class to handle Shape objects instead of just integers. The new list object is called cShapeList. The new cShapeListNode and cShapeList class declarations are shown in Listing 8-2.

LISTING 8-2 The cShapeList and cShapeListNode class declarations

```
class cShapeListNode {
friend class cShapeList;
private:
  Shape* s;
  int polyID;
  int status;
  void *pvRes;
  cShapeListNode *next;

  cShapeListNode( Shape* ps, cShapeListNode *node = NULL){ s = ps; next =
➥node; }
  cShapeListNode( Shape* ps, int pid = 0, int sFlag = 0, cShapeListNode
➥*node = NULL);
public:
  cShapeListNode* GetNext() { return (cShapeListNode*)next; }
  Shape* GetShapePtr() { return (Shape *) s; }
  int GetPolyID(){ return polyID; }
  void* GetpvResPtr() { return *pvRes; }
  void SetPolyID( int pid ) { polyID = pid; }
  int GetStatus() { return status; }
  void SetStatus( int sflag ) { status = sflag; }
};

class cShapeList {
private:
  cShapeListNode *list;
  cShapeListNode *tail;
```

```
public:
  cShapeList() { list = NULL; }
  ~cShapeList() { DeleteList(); }

  void InsertList( Shape* s );
  void InsertList( Shape* s, int pid, int sFlag );
  int  RemoveList( Shape *s );
  cShapeListNode* GetList(){ return(cShapeListNode*) list;
  void DeleteList();  // remove
};
```

The cShapeList declaration looks far more complicated than the cList class. But it isn't. The cShapeList is still performing the same functions of inserting and removing from the list. The object, cShapeListNode, is a cell node in the list object. In Chapter 7, you used the nodes to reference the data in the list. The information in each node object is a pointer to a Shape object, the asteroids, and the spaceship, as well as a polygon ID, a status field, and a special pointer. The polygon ID is used to identify what kind of shape is in the list. There are large asteroids, medium asteroids, small asteroids, a spaceship, and its missiles. You use the *polyID* field to identify these shapes. The status field will be used to determine if the object needs to be drawn or removed from the list object. The pointer *pvRes* is used to add additional data for a Shape object if it's needed.



Shape Object Definitions

A Shape object definition is the data that will be maintained by the list. A Shape object definition is an array of vertex points used to define a polygon. Using graph paper, you construct an asteroid as shown in Figure 8-1.

This is the same technique you used to define the *SpaceFoil* spaceship in Chapter 6. The asteroid in Figure 8-1 is your large asteroid. Use the *Spells of Fury* struct Point, from Chapter 6, to create a Shape object definition. Listing 8-3 shows the coordinates of the vertices defining the asteroid polygon as it would look on graph paper.

LISTING 8-3 Asteroid polygon definition

```
Point XBigRock[] = {
   {0,20},
   {20,0},
   {20,-10},
   {0,-30},
   {-10,-30},
   {0,0},
   {-20,-30},
   {-30,-10},
   {-10,0},
   {-30,0},
   {-20,20}
};
```

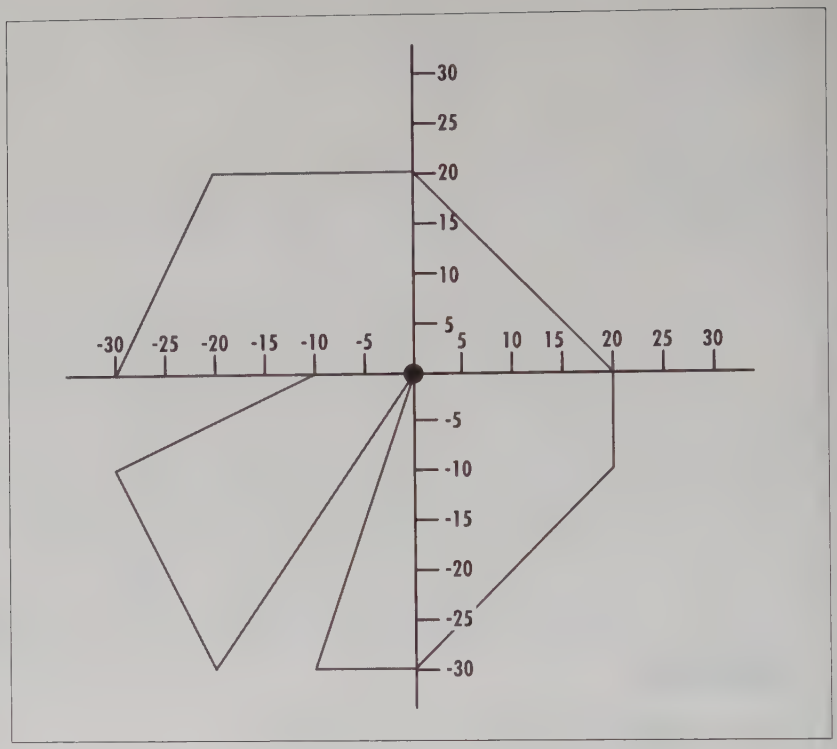


FIGURE 8-1
Cartesian point definitions of an asteroid

In Chapter 6, you used these coordinates to initialize a Shape object. In the *SpaceFoil* programming example, you stored the Shape objects in an array. In your *Asteroids* example, you will create a Shape object array, called *gameObj*, to hold your asteroid Shape object. Why are you using an array to hold your asteroid Shape, when you're suppose to be using a linked list? Simple. The *gameObj* array of Shape objects holds the Shape object master of each polygon Shape object you will use in the game. Your cShapeList object could maintain 20 asteroids at the start of the game level. But at the end, the user is advanced to the next level when he or she has destroyed all the asteroids. This means your list is empty. To initialize the next level with 35 asteroids, look up the large asteroid in the *gameObj* Shape object array and create 35 instances in your list of this one Shape object.



Initializing the *gameObj* Shape Object Array

You will take the array of Points that defines your asteroid shape and initialize the *gameObj* array entry. Currently, you only have one asteroid, so MAX_ROCK_SHAPES is set to 1. The function createDebris(), in the DEBRIS example, is used to create the Shape objects. The code for createDebris() is shown in Listing 8-4.

LISTING 8-4 Initializing the *gameObj* shape array

```
void createDebris()
{
  int size, objectSize;
  int index;

  // allocate the master shape definitions
  for (index = 0; index < MAX_ROCK_SHAPES; index++)
  {
    gameObj[index] = new Shape;
  }

  // read Point data for BIG ROCK
  size = sizeof(BigRock[0]);
  objectSize = sizeof(BigRock)/size;
  initGameObject( gameObj[BIG_ROCK_INDEX], BigRock, objectSize );
}
```

In this code, you have an array of size MAX_ROCK_SHAPES which contains one Shape definition provided in the Point array *BigRock*. The index of the object, BIG_ROCK_INDEX, is also the polygon identification, *polyID*, which you will use to describe what type of polygon is referenced in the list nodes. You can have different types of polygon objects in the list. This just identifies what type you are currently examining.

The function call, initGameObject(), was also cut and pasted from the two-player *SpaceFoil* game. This function initializes the object variables by encoding the Point array into an object and initializing the object angle and velocity attributes to 0. The function initGameObject() is shown in Listing 8-5.

LISTING 8-5 Initializing game shape objects

```
void initGameObject( Shape *s, Point *array, int arraySize )
{
  s->encodeObject(array, arraySize);
  s->setAngle(0);   // initial display angle
  s->setVector(0,0);  // initialize V = 0
}
```

Calling initGameObject(), from function createDebris(), uses the Points array *BigRock* and encodes the Points into a Shape object. The BigRock Shape object, once initialized, is stored in the *gameObj* Shape array at index BIG_ROCK_INDEX. When you initialize a game level, you will retrieve the large asteroid Shape object from gameObj[BIG_ROCK_INDEX].



Inserting a *gameObj* Entry into the Shape List

The BIG_ROCK_INDEX entry in the *gameObj* array is what you'll use to initialize the game Shape list. The function createLevel() will initialize the

asteroids in your list. The *Asteroids* game initializes the number of debris based on the game level. The function createLevel() is called by the function initGame() at the beginning of each game level. The code for initializing the game levels is shown in Listing 8-6.

LISTING 8-6 Initializing a level in *Asteroids*

```
gameLevel = 0;   // initializing the game level

int initGame()
{
  // initialize debris for level
  gameLevel++;
  debrisCount = createLevel( gameLevel );
  return TRUE;
}
```

Notice how you pass in the current *gameLevel* to use as the asteroid generation seed. The number of asteroids generated is stored in the variable debrisCount(), which is the value returned from the call createLevel(). The function createLevel() is shown in Listing 8-7.

LISTING 8-7 The createLevel function initializes the Shape list

```
int createLevel( int level )
{
  int rockcount = 2 * level + 2;
  int ix;
  double x, y;
  double vx, vy;

  for (ix = 0; ix < rockcount; ix++)
  {
    // determine a random screen position
    x = randNumGen( 0.0, 639.0 );
     y =  randNumGen( 0.0, 479.0 );
    gameObj[BIG_ROCK_INDEX]->setScreenXY(x,y);

    // determine debris direction on screen
    if ( ( x <= 320 ) && ( y <= 240 ) )
   {
    vx = BIG_ROCK_SPEED; vy = BIG_ROCK_SPEED; // DOWN RIGHT
   }

    if ( ( x <= 320 ) && ( y >= 240 ) )
    {
      vx = BIG_ROCK_SPEED; vy = -BIG_ROCK_SPEED; // UP RIGHT
    }

    if ( ( x >= 320 ) && ( y <= 240 ) )
    {
```

```
      vx = -BIG_ROCK_SPEED; vy = BIG_ROCK_SPEED; // DOWN LEFT
    }

    if ( ( x >= 320 ) && ( y <= 240 ) )
    {
      vx = -BIG_ROCK_SPEED; vy = -BIG_ROCK_SPEED; // UP LEFT
    }
    gameObj[BIG_ROCK_INDEX]->setVector( vx, vy );
    sl.InsertList( gameObj[BIG_ROCK_INDEX], BIG_ROCK_INDEX, TRUE  );
  }
  return rockcount;
}
```

The function createLevel() is responsible for generating the number of asteroids per game level and inserting the debris into the list object. The number of debris generated for the initial screen of the game level is dependent on the current game level value. The number required for the *nth* level of *Asteroids* is computed by taking the game level, doubling the value, and then adding two to it. This creates a simple geometric series based on the game level. For Level 1, there will be four asteroids; that is, 2 * (*gameLevel* =1) + 2.



Scattering the Debris on the Screen

When you generate the asteroids, you want them scattered on the game screen. In higher levels, you have no idea how many asteroids will be generated so it's a safe bet you should use a random number generator to provide *x* and *y* values for the position of the debris. The C run-time library routines in the compiler provide this feature with the call rand().

A very simple but effective random number generator can be found in the Microsoft Direct X Game SDK source code for the *Roids* example. This is a nice little utility to generate a random number in a given range of variables. The code I pillaged is shown in Listing 8-8.

LISTING 8-8 The random number generator from *Roids*

```
double randNumGen( double min, double max )
{
  double range = max - min;

  double ranNum = min + (range * (double)rand()/(double)RAND_MAX);
  return (ranNum);
}
```

For your purposes, you need a random *x* coordinate between 0 and 639 and you need a random *y* coordinate between 0 and 479. You call the function randNumGen() twice, once for the *x* coordinate of the asteroid, and the second call for its *y* coordinate. It looks like this:

```
// determine a random screen position
 x = randNumGen( 0.0, 639.0 );
```

```
y =  randNumGen( 0.0, 479.0 );
gameObj[BIG_ROCK_INDEX]->setScreenXY(x,y);
```

Each asteroid inserted into the Shape list has its own unique set of screen coordinates. Note that you assign the coordinates to the asteroid object in the *gameObj* Shape array. You do this to minimize code. Since the master image is never going anywhere on the game screen, you can use its vector and angle variables when you initialize an asteroid. You now have a shape definition of an asteroid and you have its initial screen coordinates.



Introducing Chaos to the System

If you draw the asteroids to the screen now, you will have all your asteroids moving in the same direction. This will resemble more of a flying toaster screen saver then a chaotic asteroid field. Now you must improvise a simple mechanism to introduce a little disorder to your debris field. To do this I divided the screen up into four possible coordinates as shown in Figure 8-2.

With the randomly generated *x* and *y* coordinates, you can determine the velocity vectors by filtering these coordinate values based on the quadrants the asteroid is initialized to. Listing 8-9 shows how you filter asteroid position values to initialize velocity vector components.

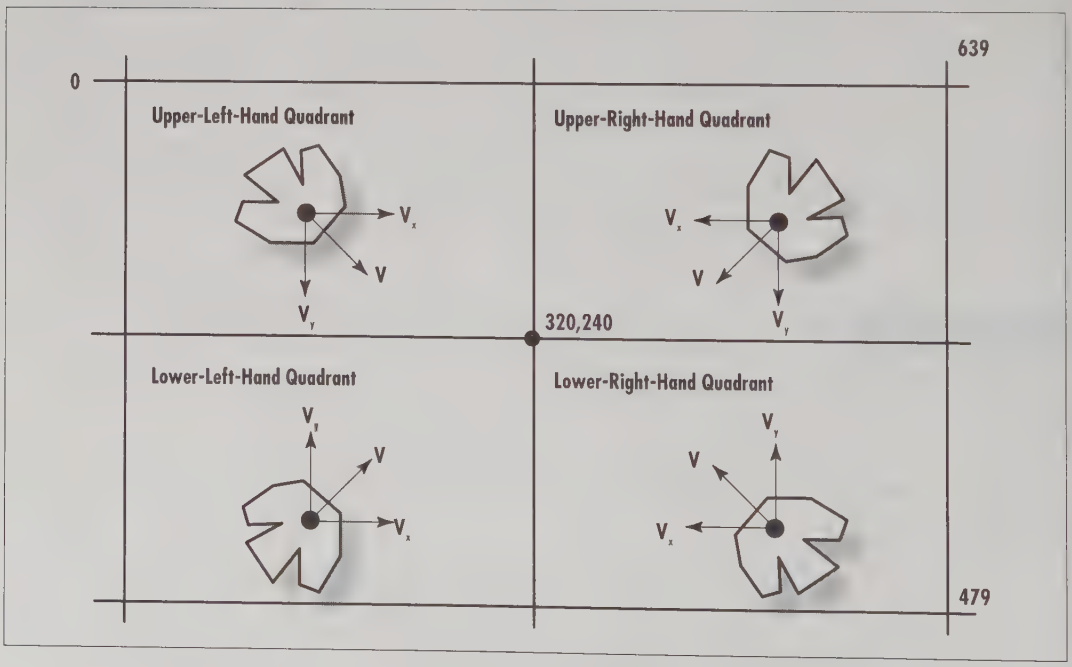


FIGURE 8-2 Asteroid velocity vectors based on screen quadrants

LISTING 8-9 Determining asteroid velocities

```
#define BIG_ROCK_SPEED 1.5
  :
// determine debris direction on screen
if ( ( x <= 320 ) && ( y <= 240 ) )
{
  vx = BIG_ROCK_SPEED; vy = BIG_ROCK_SPEED; // DOWN RIGHT
}

if ( ( x <= 320 ) && ( y > 240 ) )
{
  vx = BIG_ROCK_SPEED; vy = -BIG_ROCK_SPEED; // UP RIGHT
}

if ( ( x  > 320 ) && ( y  <= 240 ) )
{
  vx = -BIG_ROCK_SPEED; vy = BIG_ROCK_SPEED; // DOWN LEFT
}

if ( ( x  > 320 ) && ( y  > 240 ) )
{
  vx = -BIG_ROCK_SPEED; vy = -BIG_ROCK_SPEED; // UP LEFT
}
```

If the asteroid is initialized with its coordinates 0 x 320 and 0 y 240, then the asteroid is located in the upper-left-hand quadrant of the screen. For an asteroid initialized to this quadrant, you will want to initialize its velocity vector to move the asteroid in a downward direction to the right. If the asteroid's coordinates are in the range of 0 x 320 and y 240, the asteroid is located in the lower left corner of the screen. The velocity vector for this case needs to be initialized to move the debris in an upward right direction. If you have the asteroid located in the upper right corner of the screen, its coordinates are in the range of x 320 and 0 y 240. You will set the velocity vector to move the asteroid in a downward left direction. And for the last instance, an asteroid can be initialized into the lower right corner with its coordinates in the range x 320 and y 240. You will want to move the asteroid in an upward left direction. In your *Asteroids* code, you will represent the maximum width of the screen with the defined value MAX_GAME_SCREEN_X. The maximum height of the screen is the defined value MAX_GAME_SCREEN_Y.

Using quadrant checking, you can simulate nonuniform movement in your asteroid field. The direction vector is assigned by determining which quadrant the asteroid is initialized to. Shape object definition, position, and velocity are all the values you need for inserting the new asteroid object into the list. You do this with the following calls to update the object's velocity vector and the object's insertion into a list. The call to insert the asteroid into the Shape list looks like this:

```
sl.InsertList( gameObj[BIG_ROCK_INDEX], BIG_ROCK_INDEX, TRUE  );
```

Figure 8-3(a) shows the initial conditions of what a list object may look like when you insert a debris Shape object into the list. To insert into the list, the cShapeListNode constructor initializes a new node object. The gameObj[BIG_ROCK_INDEX] Shape object's data members are copied into the cShapeListNode's Shape object data member. The copy is carried out using the Shape object's overloaded Shape object assignment operator. Listing 8-10 shows the constructor for the cShapeListNode class.

LISTING 8-10 Constructor for the cShapeListNode class

```
cShapeListNode( Shape* ps, int pid, int sFlag, cShapeListNode *node)
{
  s = ps;
  polyID = pid;
  status = sFlag;
  next = node;
}
```

The Shape object is copied from the *gameObj* Shape object array, as shown in Figure 8-3(b). The cShapeListNode's Shape object now contains a snapshot of the data members of the gameObj[BIG_ROCK_INDEX] data members. Once the data members are copied the node is inserted into the list, as shown in Figure 8-3(c).



Processing the List Objects

The scheduler (see Chapter 7) is called for processing or suspending game events. If the task state is critical, the function processGame() is called to iterate the list and update the objects. Updating the objects for the *Asteroids* example, you are moving the asteroid debris to their new positions on the screen. When the list has been iterated and all objects have been updated, the game screen is refreshed with a call to the function drawDisplay().



Shape List Iteration

The list iteration is performed by the function processGame(). The function for the DEBRIS example only handles Shape objects of type BIG_ROCK_INDEX to demonstrate debris motion. This will be expanded later to handle different kinds of shapes such as the SpaceFoil, and so on. The function walks the list once each time it is called. For each Shape object on the list, it examines the Shape ID to see if it is of type BIG_ROCK_INDEX. If it is, the location of the debris on the screen is updated to the new position based on its velocity vector. The code for processGame() is provided in Listing 8-11.

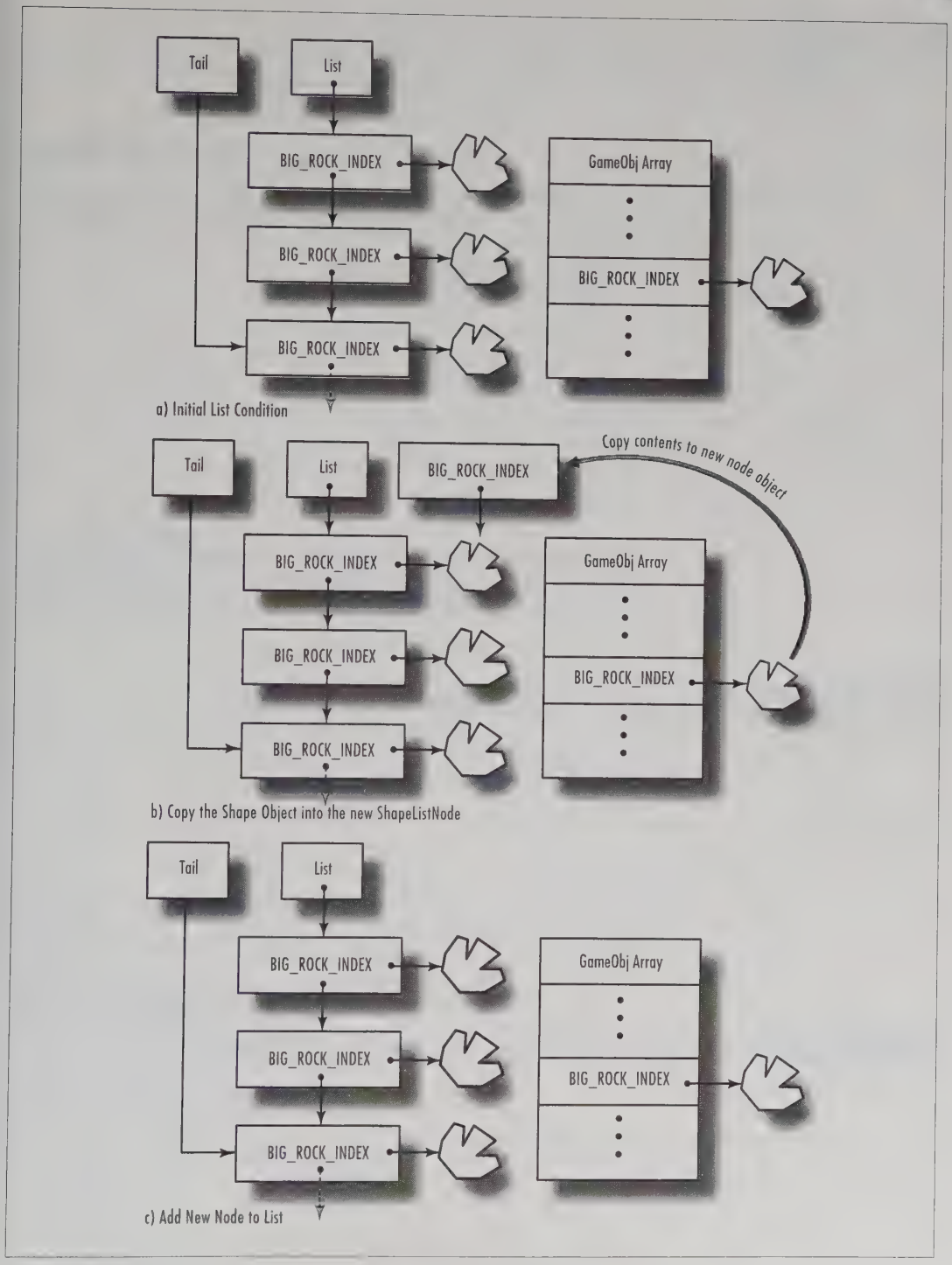


FIGURE 8-3 Inserting debris into the game list

LISTING 8-11 The processGame() function

```
int processGame( cShapeList gameList)
{
int shapeID;
int flag;
Shape *s;
for (cShapeListNode *n=gameList.GetList(); n; n=n->GetNext())
{
  shapeID = n->GetPolyID(); // get the Shape ID
  flag = n->GetStatus();    // get the object status
  s = n->GetShapePtr();     // get the Shape object ptr

  switch( shapeID )
  {
    case BIG_ROCK_INDEX:
      break;
    default:
     break;
  }
  moveObject( s );
}
```

When the list object updates are completed, the debris is then displayed with the call to drawDisplay(). The cShapeList member function, RenderList(), is called by drawDisplay() to draw the Shape objects to the screen. The code for function drawDisplay() is shown in Listing 8-12.

LISTING 8-12 Drawing the Shape objects for *Asteroids*

```
void drawDisplay()
{
  eraseScreen();
  sl.RenderList();
  flipScreen();
}
```

The cShapeList member function RenderList() iterates the list and draws the objects in the list to the screen. Shape objects in the list are drawn by calling their own member function renderShape(). Listing 8-13 provides the code for the cShapeList member function RenderList() .

LISTING 8-13 The cShapeList member function RenderList()

```
void cShapeList::RenderList()
{
  for (cShapeListNode *n=list; n; n=n->next)
  {
    n->s->renderShape();
  }
}
```

It's interesting that the amount of code needed to move the debris around is so trivial. It seems that it should be a lot harder than this but it isn't. In fact, all the work goes into defining the Shape object. Once the object is defined, it's just a matter of putting the Shape object on the list and processing the list.



The Kinematics of Chaos

In this section, I'll discuss how to bring disorder to your asteroid field. To provide a degree of realism, the space debris in *Asteroids* needs to tumble. The asteroids from the previous DEBRIS example are traveling uniformly in straight lines. You want to make your debris appear to be tumbling out of control.

The model you used to create your simple universe, in the DEBRIS example, involved randomly generating the screen *x* and *y* coordinates for each asteroid. Based on these coordinates and their positions in one of four screen quadrants, you determined the direction of motion of the debris. This simple trick provided a sense of randomness in your game but not a feeling of chaos.



The Illusion of Random Disorder

In the real universe, bodies, such as planets, asteroids, and moons, have a degree of rotation, but their tumbling is nonuniform. That is, the heavenly bodies don't all spin in the same direction. Our brains perceives this lack of uniformity as chaos. Therefore, if you can set your game objects in motion in such a manner that the debris field appears disorderly, you can provide the illusion of chaos.

Setting the space debris tumbling is really a simple task. Recall the functions rotateObjectRight() and rotateObjectLeft() you use to turn the SpaceFoil. (See Chapter 6.) The arguments you pass into these functions are pointers to Shape objects. Merely add one of these functions to processGame(), using the snippet of code shown in Listing 8-14.

LISTING 8-14 Processing Shape objects

```
switch( shapeID )
{
  case BIG_ROCK_INDEX:
  case MEDIUM_ROCK_INDEX:
  case LITTLE_ROCK_INDEX:
      rotateObjectLeft( s );
    break;
  case SPACE_FOIL_INDEX:
            :
```

continued on next page

continued from previous page

```
    break;
  default:
    break;
}
```

The addition of function rotateObject() will start your debris tumbling when it is drawn to the screen. The drawback to this approach is that by using one function to rotate all the shapes to the left, you still haven't reached a state of disorder. Again you need to introduce the element of randomness to make your debris field appear more realistic. To achieve this, you don't need to generate any more random numbers. Remember, your velocity vectors for the asteroids were set based on randomly generated coordinates. The velocity vectors in your model are constant, so they still reflect conditions from your Big Bang (the list initialization). By examining the velocity vector components of each debris object, you can assign which direction the debris will tumble. Figure 8-4 shows a diagram of how you will determine the direction of rotation. It's not obvious from the figure that rotating the objects in the upper quadrant to the right and those in the lower to the left replicate a sense of chaos. When the program executes using these rules, the effects will become immediately apparent. The idea is to fool the mind into a sense of disorder.

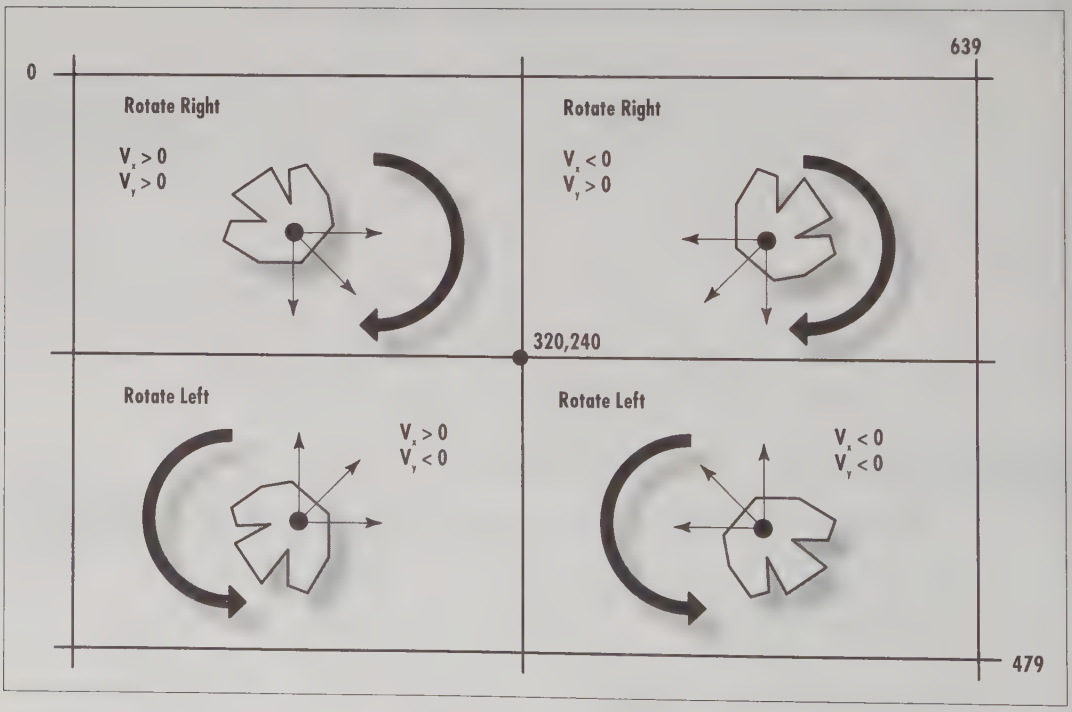


FIGURE 8-4 The first microseconds of your universe



Velocity Vectors and Quadrants

The function tumbleDebris() uses our rules of chaos to implement our universe. The function accepts a pointer to a Shape object as a calling argument. You need to retrieve the velocity vector components to determine from which quadrant the debris originated. From Figure 8-4, you see that if the object has velocity vector components $vx < 0$ and $vy > 0$, then the object originated in the upper-right-hand quadrant of the screen when the level was initialized. This object will be rotated to the right with each update by a call to function processGame(). Listing 8-15 shows the code for function tumbleDebris().

LISTING 8-15 Tumbling the space debris

```
void tumbleDebris( Shape *debris )
{
  double vx, vy;

  vx = debris->getvx();
  vy = debris->getvy();

  if ( ( vx > 0 && vy > 0) || ( vx < 0 && vy > 0 ) )
    rotateObjectRight( debris );
  else if ( ( vx > 0 && vy < 0) || ( vx < 0 && vy < 0 ) )
    rotateObjectLeft( debris );
}
```

The modifications for function processGame() to implement tumbling debris is shown in Listing 8-16.

LISTING 8-16 The processGame() function

```
int processGame( cShapeList gameList)
{
  int shapeID;
  int flag;
  Shape *s;
  for (cShapeListNode *n=gameList.GetList(); n; n=n->GetNext())
  {
    shapeID = n->GetPolyID(); // get the Shape ID
    flag = n->GetStatus();    // get the object status
    s = n->GetShapePtr();     // get the Shape object ptr

    switch( shapeID )
    {
      case BIG_ROCK_INDEX:
      case MEDIUM_ROCK_INDEX:
      case LITTLE_ROCK_INDEX:
        tumbleDebris( s );
```

continued on next page

continued from previous page

```
        break;
      case SPACE_FOIL_INDEX:
      case SF_LEFT_FLAME_INDEX:
      case SF_RIGHT_FLAME_INDEX:
      case SF_TORPEDO_INDEX:
      case SF_EXPLOSION_INDEX:
        sfNavCom( s );
        break;
      default:
        break;
    }
    moveObject( s );
  }
  return TRUE;
}
```

You have now added a simple degree of realism to your game's universe. Your asteroids tumble freely until they are hit by a missile or the pilot unwillingly slams into the space debris. At this time, the course and shape of the debris are altered.



Collisions

The object of the *Asteroids* game is to stay alive and clear the debris field. You have arrived at the point in your game implementation where you must check the position of the objects and determine if a collision has occurred. You consider a collision to occur when a Shape object touches, or nearly touches, another Shape object on the game screen. The simplest way to check for a collision is to use bounding rectangles on your Shape objects and check to see if they overlap into other objects' bounding rectangles. You must first determine how you will get a bounding rectangle from your Shape object. Remember, all of your objects are now tumbling or rotating or making hyperspace jumps. The rectangles are dynamic and must be updated when each object is moved.



Collision Detection Using Bounding Rectangles

The Shape class contains two Point arrays with the actual shapes contained in them. The *ShapeDef* is the static array you use to store the master copy of your Shape object. The sprite array data member is the dynamic image of what's been rotated and is ready to be drawn to the screen. The sprite drawn to the screen is what you want to base your bounding rectangle on for collision detection. The way you define a sprite bounding rectangle is to determine the four outermost points of the current sprite shape. Figure 8-5 depicts a graphical

representation of the bounding coordinates you would retrieve from a debris object. The Shape class member function, encodeSpriteRect(), determines these values for us. When the Shape object was first initialized, the number of vertices describing the shape was stored in the Shape class data member *pointEntries*.

To determine the current bounding rectangle of the sprite shape, you loop through the sprite vertices and find the smallest and largest *x* values and the smallest and largest *y* values. These four values compose the bounding rectangle of your current sprite shape. The smallest *x* and *y* values are the left and top values of the rectangle. The code for the Shape class member function encodeSpriteRect() is shown in Listing 8-17:

LISTING 8-17 The encodeSpriteRect() Shape class member function

```
void Shape::encodeSpriteRect()
{
  int index;

  right = left = sprite[0].x;
  top = bottom = sprite[0].y;

  for (index = 0; index < pointEntries; index++)
  {
    if (sprite[index].x > right)
        right = sprite[index].x;
    if (sprite[index].x < left)
        left = sprite[index].x;

    if (sprite[index].y < top)
        top = sprite[index].y;
    if (sprite[index].y > bottom)
        bottom = sprite[index].y;
  }
}
```

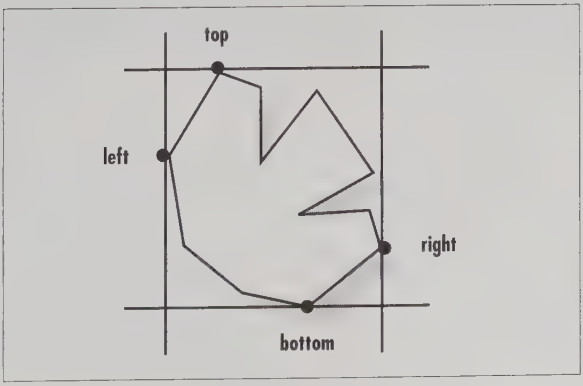


FIGURE 8-5
Examining the sprite array point entries to make a bounding rectangle

The Shape class member function, encodeSpriteRect(), is called each time the sprite has been moved to a new position on the screen. To do this, you will have to call encodeSpriteRect() every time you process game events. This means you must modify the function processGame() and add the member function call encodeSpriteRect(). As you can see, you encode the new bounding rectangle after you have moved the sprite, as shown in Listing 8-18.

LISTING 8-18 The processGame() function using encodeSpriteRect()

```
int processGame( cShapeList gameList)
{
  int shapeID;
  int flag;
  Shape *s;
  for (cShapeListNode *n=gameList.GetList(); n; n=n->GetNext())
  {
    shapeID = n->GetPolyID(); // get the Shape ID
    flag = n->GetStatus();    // get the object status
    s = n->GetShapePtr();     // get the Shape object ptr

    switch( shapeID )
    {
      // Shape object Stuff
    }
    moveObject( s );
    s->encodeSpriteRect();
  }
}
```

For debugging purposes I have included a member function in class Shape to draw the sprite's bounding rectangles. This little utility comes in handy when trying to analyze problems with sprite collisions. Figure 8-6 shows your blocky sprites in action. Listing 8-19 provides the code for Shape class member function renderSpriteRect().

LISTING 8-19 The renderSpriteRect() Shape class member function

```
void Shape::renderSpriteRect()
{

  char color = 71;
  BresenhamLine((int)left, (int)top, (int)right, (int)top, color);
  // draw left side
  BresenhamLine((int)left, (int)top, (int)left, (int)bottom, color);
  // draw right side
  BresenhamLine((int)right, (int)top, (int)right, (int)bottom, color);
  // draw top line
  BresenhamLine((int)left, (int)bottom, (int)right, (int)bottom, color);
}
```



FIGURE 8-6

Screenshot of shapes and their dynamic bounding rectangles

The member function, renderSpriteRect(), can be called from the game function renderShapeList(). As an example, you could use the function renderShapeList() shown in Listing 8-20.

LISTING 8-20 The *Asteroids* game function renderShapeList()

```
void renderShapeList( cShapeList gameList)
{
  int shapeID;
  int flag;
  Shape *s;
  for (cShapeListNode *n=gameList.GetList(); n; n=n->GetNext())
  {
    shapeID = n->GetPolyID(); // get the Shape ID
    flag = n->GetStatus();    // get the object status
    s = n->GetShapePtr();     // get the Shape object ptr
    switch( shapeID )
    {
      case BIG_ROCK_INDEX:
      case MEDIUM_ROCK_INDEX:
      case LITTLE_ROCK_INDEX:
        s->renderShape();
        s->renderSpriteRect();
        break;
```

continued on next page

continued from previous page

```
      case SPACE_FOIL_INDEX:
        s->renderShape();
        s->renderSpriteRect();
        break;
      default:
        s->renderShape();
        break;
    } // switch
  } // for
}
```

Each of the Shape objects you are drawing to the screen now has a bounding rectangle assigned to it. These rectangles are going to indicate whether you have a potential collision or not. A potential collision occurs when one sprite's bounding rectangle intersects the bounding rectangle of another sprite. If this case is true, you have a potential collision. Why do you call it a potential collision? Figure 8-7(a) shows a potential collision between two similar objects. You know from playing the classic Atari *Asteroids* that the debris do not collide with each other and split. The debris seem to move through each other transparently. This is a case of a potential collision that you do not act on. Another case is presented in Figure 8-7(b). Here you have the bounding rectangles of the SpaceFoil and a debris object intersecting. This is a valid collision and your code must respond to the event. In the program example COLLIDE, the SpaceFoil will collide with the debris object, causing it to split. However, in your real game, the SpaceFoil will explode and the debris will split in two.

Because the bounding rectangles are attributes of the Shape object, you can completely encapsulate this functionality to allow the object to check for collisions within these rectangles. The Shape class member function, CheckCollision(), examines its own bounding rectangle against another Shape object's bounding rectangle. If the two rectangles intersect, a potential collision has occurred. The code for the Shape class member function CheckCollision() is shown in Listing 8-21.

LISTING 8-21 The CheckCollision() Shape class member function

```
int Shape::CheckCollision( const Shape *s )
{

  // test if object's left-top rect is in shape rect
  if ( (left > s->left) && (left < s->right)
       && (top > s->top) && (top < s->bottom) )
    return TRUE;

  // test if object's right-top rect is in shape rect
  if ( (right > s->left) && (right < s->right)
       && (top > s->top) && (top < s->bottom) )
    return TRUE;
```




```
  // test if object's left-bottom rect is in shape rect
  if ( (left > s->left) && (left < s->right)
       && (bottom > s->top) && (bottom < s->bottom) )
    return TRUE;

  // test if object's right-bottom rect is in shape rect
  if ( (right > s->left) && (right < s->right)
       && (bottom > s->top) && (bottom < s->bottom) )
    return TRUE;
  return FALSE;
}
```

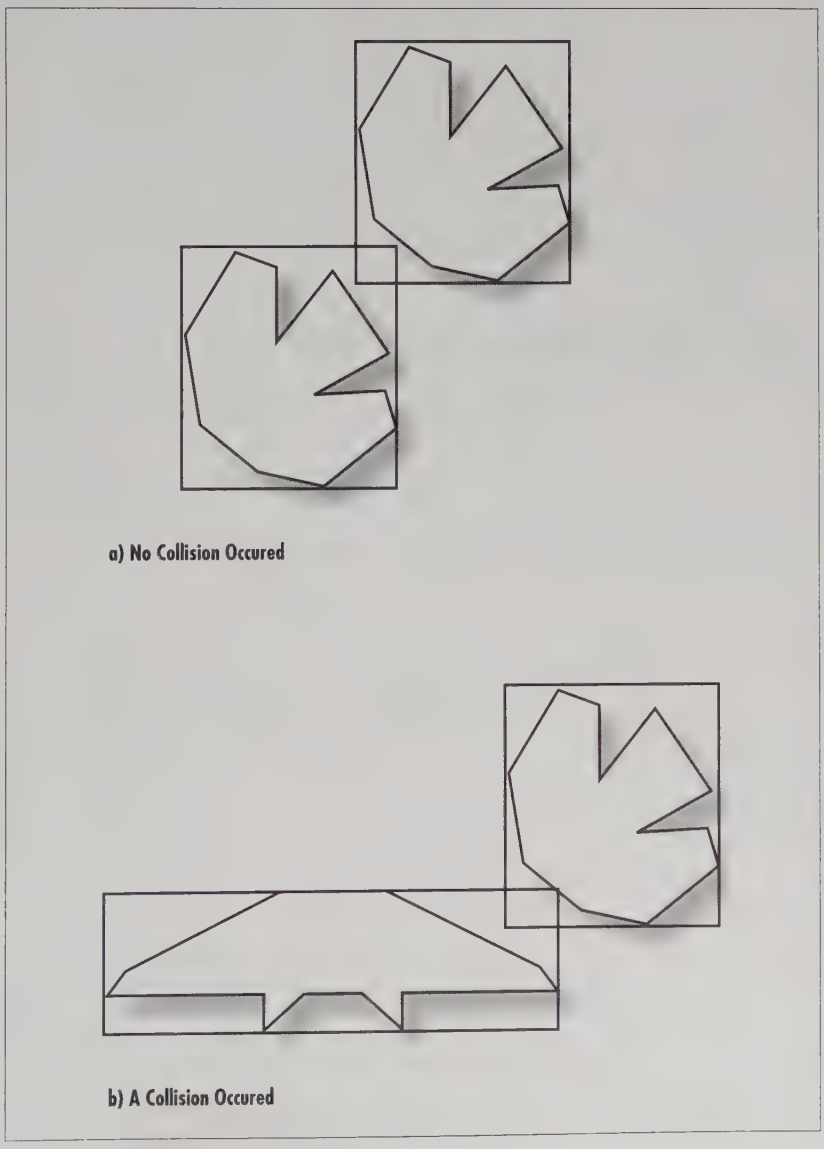


FIGURE 8-7

Checking for potential collisions

The function CheckCollision() determines if your current Shape's bounding rectangle intersects another Shape object's bounding rectangle. For instance, let your current Shape be the SpaceFoil Shape object and you are testing for a collision with a debris Shape object. Your test conditions are represented in Figure 8-8. Here you see the SpaceFoil object with its bounding rectangle's upper left coordinate represented by the *this->left* and *this->top* implicit pointer. The implicit *this* pointer indicates to you that the SpaceFoil Shape object is the calling object of function checkCollision(). The debris Shape object is passed in as a pointer. Its bounding rectangle is referenced as *s->left*, *s->top*, *s->right*, and *s->bottom*. You can see from Figure 8-8 that the debris rectangle's upper left corner overlaps the SpaceFoil's lower right corner of its bounding rectangle. In this event you have two intersecting rectangles and you return the Boolean value TRUE to indicate a collision has occurred. Actually, as you'll see in a moment, you are red-flagging that a potential collision has occurred. Glancing at Figure 8-8, you can see one of the inefficiencies in rectangle-based collision checking. The two polygons are not actually touching, yet you are calling this a collision by regional proximity. The rules governing your minute universe dictate that two bounding rectangles intersecting indicate a potential collision.



Minimizing Collision Detection

A design factor you need to take into account is which Shape objects need to check for a collision. The space debris in *Asteroids* tumble without colliding into

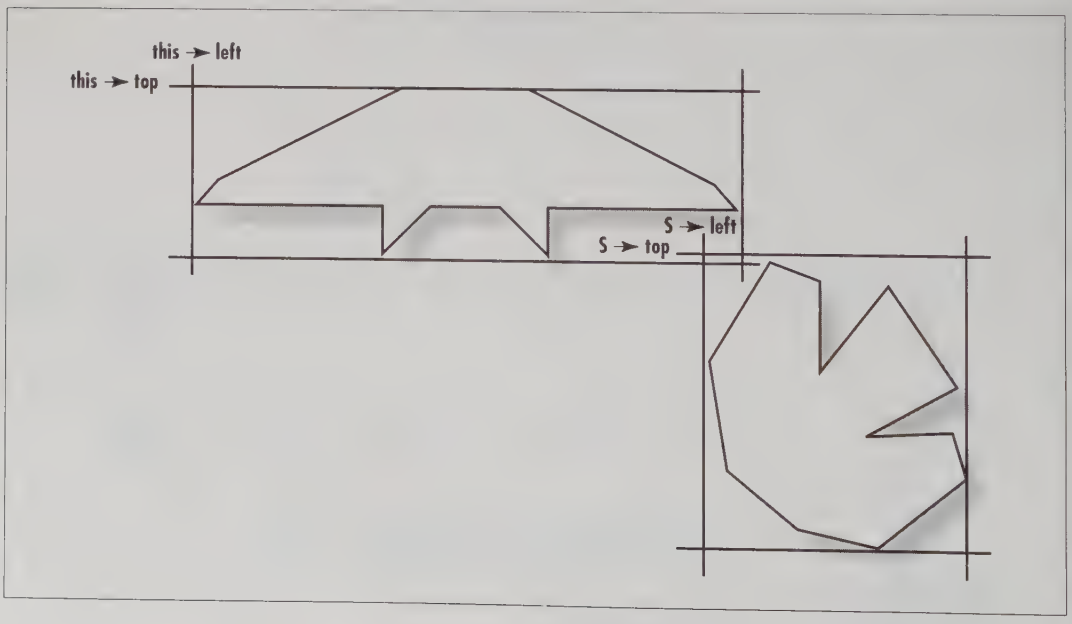


FIGURE 8-8 Collision detection using rectangle intersections

other debris. The rocks just pass through each other as if they were transparent to one another. It's a totally different story from the SpaceFoil's cockpit. The entire ship vaporizes if it slams into debris. From your game design perspective, the SpaceFoil would be the ideal candidate to perform the collision detections. Having only one Shape object checking for collisions also minimizes the load on the CPU, which helps the game run smoother.

The function *sfCheckCollision* iterates the game list and looks for colliding space debris. If the current object being examined on the list has a *polyID* of BIG_ROCK_INDEX, MEDIUM_ROCK_INDEX, or LITTLE_ROCK_INDEX, you pass this object to the Shape class member function CheckCollision(), as follows:

```
bSpaceFoilCollision = sf->CheckCollision( debris );
```

The SpaceFoil object pointer is labeled *sf* in this example. The flag *bSpaceFoilCollision* is set to TRUE in the event the ship has slammed into space debris. If *bSpaceFoilCollision* is set to TRUE, you mark the object you ran into with a status ID of REMOVE_SHAPE_INDEX. This signals the function updateDebrisCollision() as to what exactly you need to do with the debris object when the collision has occurred. For instance, when you collide with a debris of type BIG_ROCK_INDEX, your SpaceFoil explodes, and the BIG_ROCK_INDEX is removed from the list. In its place two MEDIUM_ROCK_INDEX objects are inserted into the list. This is how you simulate an asteroid splitting when you collide with it. The code for updateDebrisCollision() is shown in Listing 8-22.

LISTING 8-22 The updateDebrisCollision() function

```
void updateDebrisCollision( Shape *debris, int rockType )
{
  double vx, vy, sx, sy, dvx1, dvy1, dvx2, dvy2;
  int angle;

  // get the rotation angle
  angle = debris->getShapeAngle();

  // get screen position of debris
  sx = debris->getScreenX();
  sy = debris->getScreenY();

  // get velocity vector of debris
  vx = debris->getvx();
  vy = debris->getvy();

  setDebrisVectors( vx, vy, &dvx1, &dvy1, &dvx2, &dvy2 );
  switch( rockType )
  {
    case BIG_ROCK_INDEX:
      debrisCount--;   // remove one large debris
```

continued on next page

continued from previous page

```
      gameObj[MEDIUM_ROCK_INDEX]->setScreenXY(sx,sy);
      gameObj[MEDIUM_ROCK_INDEX]->setVector(dvx1,dvy1);
      gameObj[MEDIUM_ROCK_INDEX]->setAngle(angle);
      debrisCount++;  // add one medium debris
      sl.InsertList(gameObj[MEDIUM_ROCK_INDEX],
                    MEDIUM_ROCK_INDEX,TRUE);
      gameObj[MEDIUM_ROCK_INDEX]->setScreenXY(sx,sy);
      gameObj[MEDIUM_ROCK_INDEX]->setVector(dvx2,dvy2);
      gameObj[MEDIUM_ROCK_INDEX]->setAngle(angle);
      debrisCount++;  // add one medium debris
      sl.InsertList(gameObj[MEDIUM_ROCK_INDEX],
                    MEDIUM_ROCK_INDEX,TRUE);
      break;
    case MEDIUM_ROCK_INDEX:
      debrisCount--;
      gameObj[LITTLE_ROCK_INDEX]->setScreenXY(sx,sy);
      gameObj[LITTLE_ROCK_INDEX]->setVector(dvx1,dvy1);
      gameObj[LITTLE_ROCK_INDEX]->setAngle(angle);
      debrisCount++;  // add one little debris
      sl.InsertList(gameObj[LITTLE_ROCK_INDEX],
                    LITTLE_ROCK_INDEX,TRUE);
      gameObj[LITTLE_ROCK_INDEX]->setScreenXY(sx,sy);
      gameObj[LITTLE_ROCK_INDEX]->setVector(dvx2,dvy2);
      gameObj[LITTLE_ROCK_INDEX]->setAngle(angle);
      debrisCount++;  // add one little debris
      sl.InsertList(gameObj[LITTLE_ROCK_INDEX],
                    LITTLE_ROCK_INDEX,TRUE);
      break;
    case LITTLE_ROCK_INDEX:
      debrisCount--;
      break;
    default:
      break;
  }
}
```



Imitating Elastic Collisions

A key operation that updateDebrisCollision() performs is computing the new velocity vectors when the debris splits. According to the television show *Beakman's World* on Saturday mornings, when a billiard cue ball strikes a stationary eight ball head-on, the two balls momentarily stick and then move off in their own angles of deflection. The classic pool ball collision is known in physics as an elastic collision. The angles of deflection of the two objects are directly related to the angle and velocity of the striking cue ball. Figure 8-9 shows our shady pool hall example of the colliding pool balls.

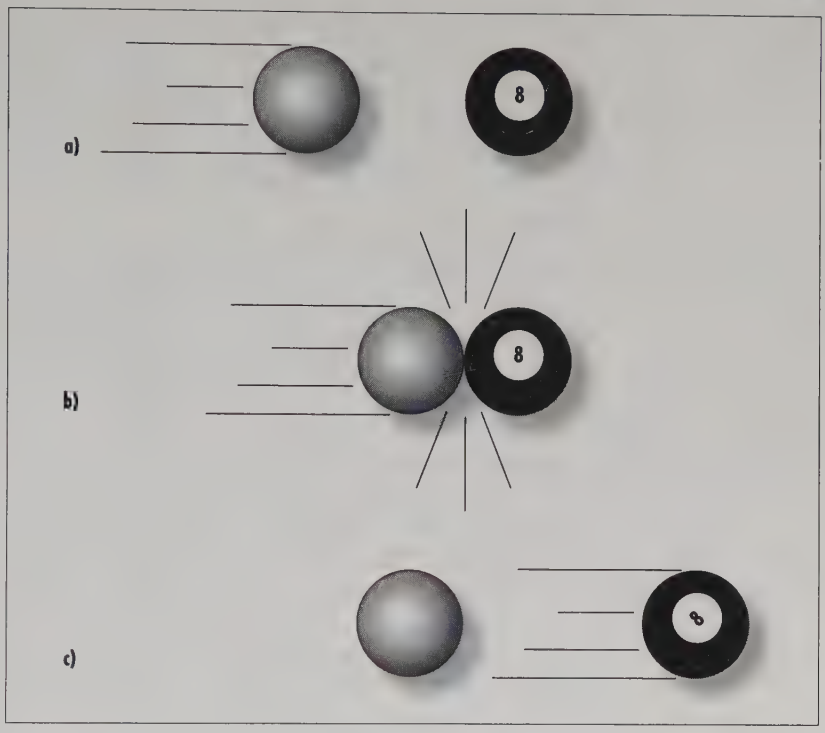


FIGURE 8-9

The physics of colliding pool balls

Figure 8-9(a) shows the initial conditions of the two balls on the billiard table. For you purists out there, note that the two balls have the same mass. The cue ball is moving towards the stationary eight ball at a velocity of *V*. In Figure 8-9(b) you see the moment of collision, and the two bodies momentarily meet. The colliding body, the cue ball, comes to a stop and the struck eight ball moves off, as shown in Figure 8-9(c), with the same speed and direction that the cue ball had. This is the simple model for the *Asteroids* universe which you will use in your game. Now, obviously, there are all sorts of rules governing collisions. For instance, in reality, the SpaceFoil is much smaller in mass than a tumbling debris of rock in space. You need to simplify things and only provide the illusion of collision. You can do this by borrowing a few rules from math and physics and waving your programmer's magic wand at the rest of the computations.



The Angle of Deflection

Your game will assume that all objects are of equal masses. This alone immediately minimizes the number of equations on the scratch pad. Secondly, you will brute force the angle of deflection during a collision. The information

you will be using for collisions will be retrieved from the debris object's velocity vectors. The immediate assumption is that the SpaceFoil will explode on collision so you will not need any values from this Shape object.

Since you are the all-powerful creator of your little game universe, you can create your own rules for collision. As long as you don't put an asteroid on the screen that looks like an eight ball, no one will know that your universe behaves like balls on a pool table.

When a collision occurs, the larger space debris appears to be split in half. In actuality, you insert two smaller rocks travelling in the deflected direction into the list. The old debris object is removed from the list. The function updateDebrisCollision() extracts the values you need to insert two new smaller debris objects onto the list. The code for this is as follows:

```
vx = debris->getvx();
vy = debris->getvy();
setDebrisVectors( vx, vy, &dvx1, &dvy1, &dvx2, &dvy2 );
```

These are the velocity vectors of the old debris object that was involved in the collision with the SpaceFoil. These two values provide you with all the information you need to compute the deflection velocity vectors of the two new debris objects you will insert in the old object's place on the screen. The new velocity vectors are set by the setDebrisVectors() function. You know from physics that the velocity components *vx* and *vy*, which you stored in the Shape object, tell you the angle that the object is traveling. In function setDebrisVector(), you calculate this angle, called *theta* in the COLLIDE example code, by taking the arctangent of *vy* divided by *vx*. The angle *theta* is the direction on the screen that your original debris was heading before the collision. In your code you have this:

```
theta = atan(vy/vx);
```

You examine the debris heading and set constants *kx* and *ky*. These values represent whether *vx* and *vy* were positive or negative vector components. The constants *kx* and *ky* will be used to deflect your two new debris objects in the opposite direction of the original object involved in the collision. The values these constants can assume are -1 and 1 to reflect the vector components' direction. You determine the direction of the vector and the recoil constants using the code shown in Listing 8-23.

LISTING 8-23 The space debris vector recoil constants

```
kx = 1;
if (vx < 0)
  kx = -1;

ky = 1;
if (vy < 0)
  ky = -1;
```

This provides you with enough information to calculate the two new vectors to be used by the debris when the rock splits in half. Using your high school knowledge of trigonometry, you calculate the magnitude of the velocity vector using the vector components. You compute the magnitude of the velocity vector with the following line of code in updateDebrisCollision():

```
vector = sqrt(vx*vx + vy*vy);
```

The variable *vector* is the magnitude of your velocity from the original debris in the collision. Now, unlike the billiard ball example, your debris splits in two during a collision. This means you have to provide for some sense of realism when the object splits and these two new debris objects move off in the asteroid field. Figure 8-10(a) shows your asteroid instantaneously at the moment of collision. You see its velocity vector components and you see the object has not yet split in two pieces. In Figure 8-10(b), you see two new sets of velocity vector components being generated from the original object in the collision. The two new vectors are actually the original vector changed to a new heading of angle *theta* ± angle *alpha*. The angle *alpha* is a constant set to 30 degrees. The new velocity vectors are set to ±30 degrees the original vector. Listing 8-24 shows how you initially generate vectors for the new space debris.

LISTING 8-24 Initializing velocity vectors of space debris

```
alpha1 = 30 * RADIAN_UNIT;  // new debris travels +-30 degrees
alpha2 = - alpha1;          //  from old vector
alpha1 += theta;                  // set new rock vector angle
*dvx1 = vector * cos(alpha1) * kx;// calculate new x component
*dvy1 = vector * sin(alpha1) * ky;// calculate new y component
alpha2 += theta;                  // set new rock vector angle
*dvx2 = vector * cos(alpha2) * kx;// calculate new x component
*dvy2 = vector * sin(alpha2) * ky;// calculate new y component
```

The angles *alpha1* and *alpha2* are your 30-degree offsets expressed in radian units. To send the new debris off in a direction that makes it appear as if it broke off from the original, you add the angles *alpha1* and *alpha2* to the original angle *theta*. This results in two new velocity vectors as shown in Figure 8-10(b). Using your billiard ball model, you want the debris to move in the opposite direction of impact. To simplify this computation, the new debris moves off in the direction opposite of the original debris. Multiplying the two new velocity vector components by *kx* and *ky* sends your debris off in the opposite direction, as shown in Figure 8-10(c). This isn't how one would truly model a collision if one were an astrophysicist and had a Cray at one's disposal, but it serves the game programmer's purposes and it's simple.

To initialize the first new debris object, you add the angle offset *alpha1* to the original angle *theta*. The sum of these angles is stored in the variable *alpha1*. Your new velocity components for this new debris object are these:

vx1 = |V| cos(alpha1)

vy1 = |V| sin(alpha1)

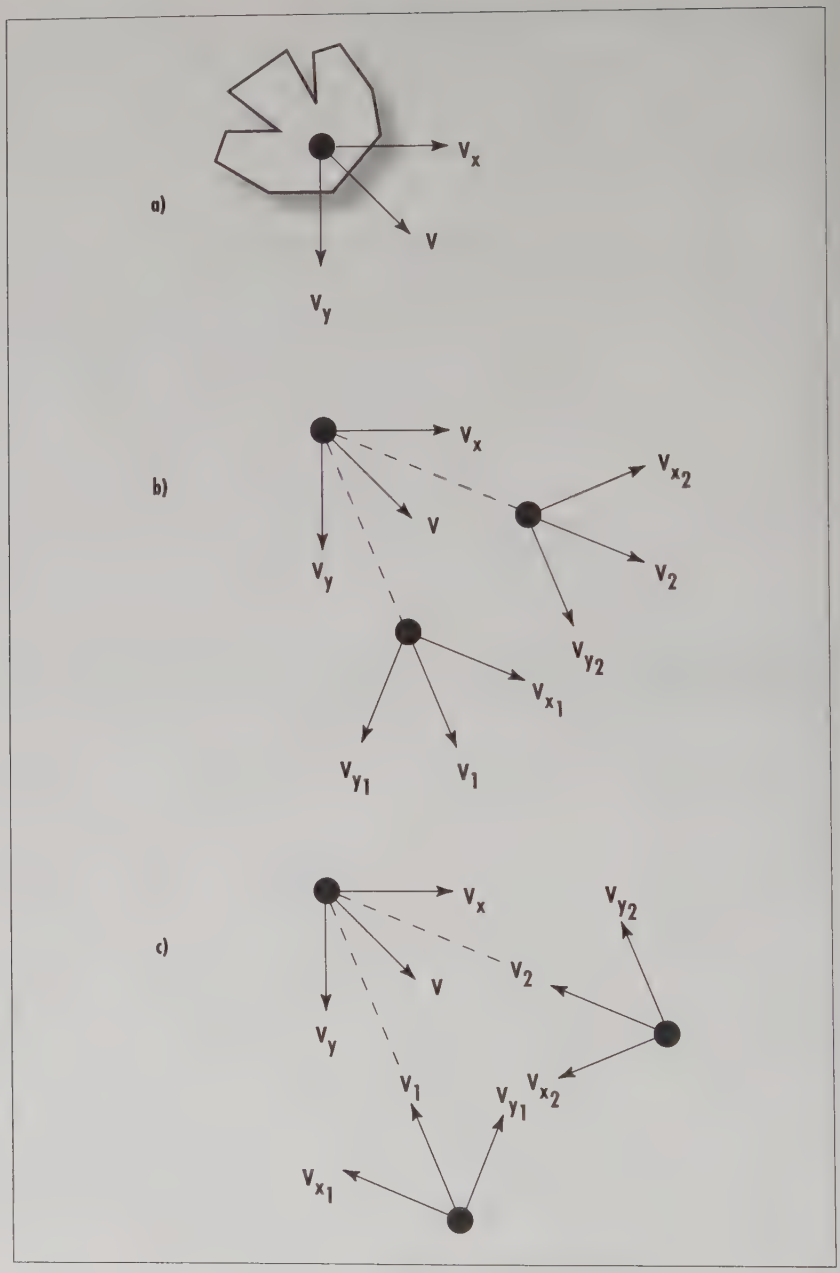


FIGURE 8-10
Determining velocity vectors for new rock chunks

The code in setDebrisVectors() that handles the new velocity components is provided in Listing 8-25.

LISTING 8-25 Computing the new velocity components of space debris

```
alpha1 += theta;              // set new rock vector angle
*dvx1 = vector * cos(alpha1) * kx;// calculate new x component
*dvy1 = vector * sin(alpha1) * ky;// calculate new y component
```

The computations and code to compute the second velocity vector components for the second debris object are identical. The only difference is that angle *alpha2* is angle *theta* minus 30 degrees, instead of adding as you did in the previous case. The source for setDebrisVectors()is shown in Listing 8-26.

LISTING 8-26 The setDebrisVectors() function

```
void setDebrisVectors(double vx, double vy, double *dvx1, double *dvy1,
➥double *dvx2, double *dvy2)
{
  double theta, alpha1, alpha2;
  double vector;
  int kx, ky;

  // directional recoil constant
  kx = 1;
  if (vx < 0)
    kx = -1;

  ky = 1;
  if (vy < 0)
    ky = -1;

  vector = sqrt(vx*vx + vy*vy);
  theta = atan(vy/vx);              // get angle of velocity vector
  alpha1 = 30 * RADIAN_UNIT; // new debris travels +-30 degrees
  alpha2 = - alpha1;                //  from old vector

  alpha1 += theta;             // set new rock vector angle
  *dvx1 = vector * cos(alpha1) * kx;// calculate new x component
  *dvy1 = vector * sin(alpha1) * ky;// calculate new y component

  alpha2 += theta;             // set new rock vector angle
  *dvx2 = vector * cos(alpha2) * kx;// calculate new x component
  *dvy2 = vector * sin(alpha2) * ky;// calculate new y component

}
```

Returning from the function setDebrisVectors(), you have the velocity components for two new pieces of debris that you need to insert into the list in place of the original debris that split in half. You now need to set the object angles, screen positions, and velocity vectors of the new objects. This is all handled in the function updateDebrisCollision() in a *switch* statement, as shown in Listing 8-27.

LISTING 8-27 Updating the new space debris objects for list insertion

```
switch( rockType )
{
  case BIG_ROCK_INDEX:
    debrisCount--;   // remove one large debris
    gameObj[MEDIUM_ROCK_INDEX]->setScreenXY(sx,sy);
    gameObj[MEDIUM_ROCK_INDEX]->setVector(dvx1,dvy1);
    gameObj[MEDIUM_ROCK_INDEX]->setAngle(angle);
    debrisCount++;  // add one medium debris
    sl.InsertList(gameObj[MEDIUM_ROCK_INDEX],MEDIUM_ROCK_INDEX,TRUE);
    gameObj[MEDIUM_ROCK_INDEX]->setScreenXY(sx,sy);
    gameObj[MEDIUM_ROCK_INDEX]->setVector(dvx2,dvy2);
    gameObj[MEDIUM_ROCK_INDEX]->setAngle(angle);
    debrisCount++;  // add one medium debris
    sl.InsertList(gameObj[MEDIUM_ROCK_INDEX],MEDIUM_ROCK_INDEX,TRUE);
    break;
  case MEDIUM_ROCK_INDEX:
    debrisCount--;
    gameObj[LITTLE_ROCK_INDEX]->setScreenXY(sx,sy);
    gameObj[LITTLE_ROCK_INDEX]->setVector(dvx1,dvy1);
    gameObj[LITTLE_ROCK_INDEX]->setAngle(angle);
    debrisCount++;  // add one little debris
    sl.InsertList(gameObj[LITTLE_ROCK_INDEX],LITTLE_ROCK_INDEX,TRUE);
    gameObj[LITTLE_ROCK_INDEX]->setScreenXY(sx,sy);
    gameObj[LITTLE_ROCK_INDEX]->setVector(dvx2,dvy2);
    gameObj[LITTLE_ROCK_INDEX]->setAngle(angle);
    debrisCount++;  // add one little debris
    sl.InsertList(gameObj[LITTLE_ROCK_INDEX],LITTLE_ROCK_INDEX,TRUE);
    break;
  case LITTLE_ROCK_INDEX:
    debrisCount--;
    break;
  default:
    break;
}
```

The screen *x* and *y* coordinates and the angle of the debris were all retrieved earlier with the velocity components before the call to the function setDebrisVectors() was made. With this information and the new velocity vectors, you can insert two new debris objects into the list. The insertion of the debris looks just like the code you used to initialize the list with debris. Assuming that the SpaceFoil just collided with a chunk of debris of type BIG_ROCK_INDEX, the *case* statement would perform the following operations to insert two new debris objects of type MEDIUM_ROCK_INDEX onto the list, as shown in Listing 8-28.

LISTING 8-28 Handling a collision with shape of type BIG_ROCK_INDEX

```
case BIG_ROCK_INDEX:
    debrisCount--;   // remove one large debris
    gameObj[MEDIUM_ROCK_INDEX]->setScreenXY(sx,sy);
    gameObj[MEDIUM_ROCK_INDEX]->setVector(dvx1,dvy1);
    gameObj[MEDIUM_ROCK_INDEX]->setAngle(angle);
    debrisCount++;  // add one medium debris
    sl.InsertList(gameObj[MEDIUM_ROCK_INDEX],MEDIUM_ROCK_INDEX,TRUE);
    gameObj[MEDIUM_ROCK_INDEX]->setScreenXY(sx,sy);
    gameObj[MEDIUM_ROCK_INDEX]->setVector(dvx2,dvy2);
    gameObj[MEDIUM_ROCK_INDEX]->setAngle(angle);
    debrisCount++;  // add one medium debris
    sl.InsertList(gameObj[MEDIUM_ROCK_INDEX],MEDIUM_ROCK_INDEX,TRUE);
    break;
```

Here you decrement the count of debris by one. This is the action you are taking for bookkeeping. You remove one debris object. In this case, it's a BIG_ROCK_INDEX type object. With each debris object you add to the list, you increment the *debrisCount*. As you can see from your code, the two MEDIUM_ROCK_INDEX objects increment the *debrisCount* twice. The *debrisCount* is used in the game to determine if the asteroid field has been cleared for level *n*. So you want to accurately maintain what debris is being inserted and removed from the list.

A key point to notice is that you haven't actually removed anything from the list. You have just marked objects you want removed and gone on from there. You did this because it's possible to collide with more than one Shape object at one time. Remember, the debris can move through each other transparently. What if the SpaceFoil collides during this particular event? How do you know which asteroid you collided with, if not both? Marking the debris to remove is the simplest way to handle this situation. You can collide with multiple debris and just iterate the list, once the collision updates are completed, to remove all the tagged debris. You do this in your game in processGameEvents(). The function processGame() that is used in the COLLIDE example is presented in Listing 8-29.

LISTING 8-29 The processGame() used in the COLLIDE example function

```
int processGame( cShapeList gameList)
{
  int shapeID;
  int flag;
```

continued on next page

continued from previous page

```
  Shape *s;
  for (cShapeListNode *n=gameList.GetList(); n; n=n->GetNext())
  {
    shapeID = n->GetPolyID(); // get the Shape ID
    flag = n->GetStatus();   // get the object status
    s = n->GetShapePtr();    // get the Shape object ptr

    switch( shapeID )
    {
      case BIG_ROCK_INDEX:
      case MEDIUM_ROCK_INDEX:
      case LITTLE_ROCK_INDEX:
        tumbleDebris( s );
        break;
      case SPACE_FOIL_INDEX:
      case SF_LEFT_FLAME_INDEX:
      case SF_RIGHT_FLAME_INDEX:
      case SF_TORPEDO_INDEX:
      case SF_EXPLOSION_INDEX:
        sfNavCom( s );
        break;
      default:
        break;
    }
    moveObject( s );
    s->encodeSpriteRect();
  }
  sfCheckCollision(gameList);
  sl.RemoveList(REMOVE_SHAPE_INDEX);
  return TRUE;
}
```

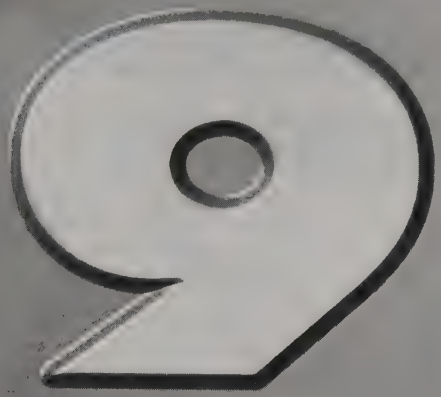
The collision checking for the SpaceFoil is handled first by function sfCheckCollision(), and then you perform the object removal with the list member function RemoveList(). These two functions are how you handle debris collisions in the *Asteroids* game. Other types of collision checking exist and operate more efficiently than rectangular bounds checking. One particular source code I retrieved from the Net at **sunsite.com** used collision detection based on whether line segments of the object polygons intersected. This is a very accurate method for handling polygon collision detection. However, UNIX platform workstations still tend to have just a tad more floating point computing power than microcomputers. This inadequacy in PC computing will be fading soon, I suspect.



Concluding the Chaos

As you may have seen, there is a lot to the *Asteroids* game. It has a simple appeal, which is probably why it has been so popular for the past 15 years. A really interesting port of *Asteroids* in the past three years has been for the Macintosh. It's a shareware program named *Maelstrom*. I was excited to participate in the developer's beta testing of the program. I hope the game designer gets his hands on a copy of the Microsoft Direct X Game SDK and ports *Maelstrom* over to Windows 95. It's got the same high-paced edge as the game *Raptor*.

You can take the concepts presented here and build your own list of *Asteroids* sprites. You can use the collision techniques presented and use them for sprites as well. So play around with the *Spells of Fury* code and create a clever new twist to *Asteroids* like the developer of *Maelstrom* did.



BUBONIC THE WARTHOG

BUBONIC THE WARTHOG

The Asteroids game was a nice stroll down memory lane. But now it's time to write *Sega, Super Nintendo,* and *Sony PlayStation* console-style games. With a game display of 640 x 480 pixels, palettes of 256 colors or better, and phenomenal frame-rates of 45 to 50 frames per second, you are ready to design games for the PC that exceed the capabilities of the game consoles. In this chapter, you'll develop a tile-based game with a virtual terrain called *Bubonic the WartHog*. You can take a clever guess at what console game you're going to design using the Microsoft Windows 95 Direct X Game SDK.

EPISODE THE SPHINX



When I opened the door leading out of the core memory I knew I would regret it. It was a gut feeling.

"What are you waiting for? The central processing center is just past the chamber directly ahead of you." 13 prodded eagerly.

"That's easy for you to say. It's not your seat that's in the sling here." I replied back.

"You've got a full arsenal from the core memory. You're ready to kick some..." 13's excitement was interrupted by the roar of a very big lion.

"Oh boy..." I muttered. "I don't mind the mummies, or the giant statues but I do not want to be somebody or something's lunch." Going against my better judgment, I stepped into the new chamber. A large sandstone door dropped and sealed me in with no means of getting back to the safety of the hallway.

"Welcome, worthy opponent. You have come here to die," a deep voice echoed. Without any doubt, I knew it was the voice of Agamemnon. In front of me a large bright blue ball flashed, and then disappeared. A giant sphinx-like creature, with the lower body of a lion and the torso of a man, stood before me.

"Ready? Fight!" Agamemnon ordered.

"Oh hey! I know this game!" 13 eagerly informed me. "Hold on. I'll hook up my gamepad and get you through it."

"13, that's not such a good idea. Just get me out of here!"

The sphinx stepped over towards me. I felt like I was in an old Sinbad movie with the Claymation monsters. The monster reached down and picked me up by my ankles and threw me into the wall. My head was spinning and I tried to focus my eyes.

Involuntarily, I did a Jackie Chan-style ninja backflip and kicked the monster in the torso about nine times in mid-air before landing on my feet.

"This is great! Did you see that?" 13 commented on his own gaming skills.

"13, you're making me sick. My head is dizzy."

"Hang on. I managed to knock the sphinx's health down," 13 reported.

"My health isn't too good right now. I see six monsters."

"Hang in there, Bary. There's only one sphinx."

13 flung me at the beast again. With another act of inhuman gymnastics, my body took to the air. The sphinx spun around and knocked me down with a brutal strike from its tail. 13 commanded my body up from the floor, using his gamepad.

"Hey, let's see if any button combinations give you special moves!" 13 exclaimed, forewarning me of his experiment.

He flung me around like a marionette on drugs. My body, under 13's command, leaped in front of the sphinx, and I raised my hands into the air. Lightning shot from my hands, striking the monster. The monster froze in its tracks, paralyzed. 13 ran me into the creature, striking it with kicks and lethal punches. He was obviously having the time of his life.

"Bary, the health bar on my display shows the sphinx is a goner."

The sphinx spun in a dizzy stance, sort of like me when the creature first flung me into the wall. 13 drove the last lethal blow and the creature vanished.

"Bary wins," Agamemnon commented. The door leading to the central processing center opened.

"We did it, Bary. We're lean mean fighting machines!" 13 was very excited by his accomplishment.

"13, unplug your gamepad. I really just want to sit down!"



A World Made of Tiles

Most console-style games, such as Sega's *Sonic the Hedgehog*, Nintendo's *Super Mario Brothers*, and Sega's *X-Men*, require the player to progress in a linear

fashion to complete a level. The game is completed when the player successfully finishes all the levels of the game. The levels in these games are usually composed of maps. The maps are two-dimensional and the character can travel up or down, by using ladders, stairs, or elevators, or by jumping. The player enters the level by being placed somewhere on the left-hand side of the map. The goal is for the player to make it to the right-hand side of the map to finish a level. Some excellent examples of maps can be found in video game magazines such as *GamePro*. Usually *GamePro* will take multiple screenshots of a game to produce a map. Dozens of other magazines devoted to gameplay are available on the newsstand so you'll probably be able to find one with a level map in it.

A map in a game magazine is usually composed of five screenshots from end to end. If the game allows you to travel up and down in a map, like *Sonic the Hedgehog* does, you may need to compose a collage of five screenshots for the upper layer of the map, five screenshots for middle layer of the map, and another five for the lower layer of the map. In all you have a map that requires 15 screens to compose a level to a game. Assuming you port *Sonic the Hedgehog* over to the Windows 95 Direct X Game SDK, with a resolution of 640 x 480 pixels times fifteen screens, you're talking a lot of video memory. Actually, quite a bit more memory than you have available.

The solution to this problem is to use a technique called *tiling*. Since you can't store 15 screens of your map in video memory, you will use small sprites called tiles to construct the level map. A hundred or so 32 x 32 pixel tiles can be used like a mosaic piece of art to construct a background. Exact placement of each tile is determined by a map provided by the developer.



Introduction to Tile Maps

Over the years, I have seen some very clever data structures to represent tile maps. In this discussion you'll tile a static terrain using a simple map. For this example, you'll use the *Spells of Fury* DITEST application from Chapter 4. You'll add a slight twist to this implementation and add a tile map for your character, Bubonic the WartHog, to run on. As an introductory example, your map will only comprise one screen. The screen is composed of 640 x 480 pixels and your tiles are 32 x 32 pixels each. If you provide a relationship of one tile is equivalent to 32 pixels, you come up with a logical array of 20 cells x 15 cells. This logical array is going to be your map for the screen. The map will provide a correlation of tile placements to the physical DirectDraw surface. Each entry in the map will represent a tile type. The tile types are numbered sequentially, 0 through 42 for the TILES program demonstration. Figure 9-1 shows the terrain tiles located on the offscreen DirectDraw surface, with their corresponding tile types.



FIGURE 9-1
Terrain tile types used in the TILES example program

The Terrain Map

You can create a *TerrainMap* array that is 15 rows with 20 cells per row to represent the screen. Each cell of the two-dimensional array will hold a tile. The *TerrainMap* row and cell index directly maps to the 640 x 480 flipping DirectDraw surface used in the example. The *TerrainMap* definition is shown in Listing 9-1.

LISTING 9-1 Bubonic terrain map

```
int TerrainWidth = 20;
int TerrainHeight = 15;

int TerrainMap[GAME_PORT_HEIGHT][GAME_PORT_WIDTH] = {
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        { 1, 6,11,16,21,25,29,33,37,40, 1, 6,11,16,21,25,29,33,37,40},
        { 2, 7,12,17,22,26,30,34,38,41, 2, 7,12,17,22,26,30,34,38,41},
        { 3, 8,13,18,23,27,31,35,39,42, 3, 8,13,18,23,27,31,35,39,42},
        { 4, 4, 4, 4,24,28,32,36, 4, 4, 4, 4, 4, 4,24,28,32,36, 4, 4}
};
```

The first ten rows of your terrain map consist of zeroes. You used this value to flag that you will not be placing tiles on this 32 x 32 pixel region of the flip

DirectDraw surface. Our DITEST example didn't have a background so you're going to add one here. The zeroes identify an empty space so you don't write over the background art. Starting with row 11 of your terrain map, you have non-zero values for the tile types. The tiles are labeled in Figure 9-1 with integer values. These values directly correspond to the values in the *TerrainMap* array.



The Tile Lookup Table

At the moment you have a terrain map that is 15 rows with 20 cells per row. The terrain map contains integer values of tile types stored on the offscreen DirectDraw surface. How you would go about retrieving the tiles from the offscreen DirectDraw surface? You only know in your map that tile type 36 is located somewhere on the offscreen DirectDraw surface. You need a lookup table that provides the one-to-one correlation between tile types and their location offscreen. The TILES program uses an array of the CELL structure to maintain a tile ID to coordinate mapping. The CELL structure definition is shown in Listing 9-2.

LISTING 9-2 Bubonic tile CELL structure

```
typedef struct {
  int posx;
  int posy;
} CELL;

CELL cells[] = {
       // x,  y  upper left corner of the tile
       {  0,   0},  // tile type 0
       {  1, 482},  // tile type 1
       {  1, 516},  // tile type 2
       {  1, 550},  // tile type 3
       {  1, 584},  // tile type 4
       {  1, 618},  // tile type 5
       { 35, 482},  // tile type 6
              :
       {273, 550},  // tile type 39
       {307, 482},  // tile type 40
       {307, 516},  // tile type 41
       {307, 550}   // tile type 42
};
```

The tile type is actually an index into the tile lookup table, which is the array of the CELL structure. This array contains the upper left coordinate of the tile as it is stored in the offscreen DirectDraw surface. Tile type 0 is actually a dummy place-holder. Tile type 1 is stored in the offscreen DirectDraw surface with its upper left corner at 1,482. You may want to open the BUBONIC.BMP used in the TILES program with the Microsoft Paint application found in Windows 95 Accessories. Move the mouse cursor while watching the

coordinates in the lower-left-hand corner. When you reach the point 1,482 you'll see the same tile that is labeled tile 1 in Figure 9-1. The cells array is your tile lookup table. You use the tile type as an index into the cells array to locate the upper left corner of your tile.



Rendering with Tiles

You've got a terrain map, a tile lookup table, and your tiles stored in an offscreen DirectDraw surface. Now you need to assemble your mosaic and draw it to the display. The first thing you want to do before you render the tiles is to put your static background in place. If you opened the BUBONIC.BMP with the Microsoft Paint accessory, you noticed the top 640 x 480 pixels of the 640 x 1250 DIB was a background. You want to draw your background first. Using the routine drawOneToBack() from the *Spells of Fury* DDTOOLS code, your code for drawing the background is shown in Listing 9-3.

LISTING 9-3 Drawing the Bubonic background

```
void drawBackGround()
{
  RECT rc;
  rc.left = 0;
  rc.top = 0;
  rc.right = 639;
  rc.bottom = 479;
  drawOneToBack(0,0,rc,TRUE);
}
```

The function is very simple. It is composed of a Windows RECT structure and a call to drawOneToBack(). That's all you do to get the background from the offscreen DirectDraw surface, *lpDDSOne*, to the visible flipping DirectDraw surface, *lpDDSBack*.

The idea here is that with tiles you draw in layers. The background is the first layer you draw and then you add the tiles on top of that. The tiles can also be layered. For the example program TILES, use the function drawTerrain() to render your tiles to the flipping DirectDraw surface. The code for drawTerrain() is shown in Listing 9-4.

LISTING 9-4 The drawTerrain function

```
#define GAME_PORT_HEIGHT 15
#define GAME_PORT_WIDTH 20
void drawTerrain()
{
  int   mapRow, mapIndex, tileIndex;
  int   tileSize = 32;  // 32 x 32 pixels
  RECT  srcRect;
```

```
  int   destx, desty;

  for (mapRow = 0; mapRow < GAME_PORT_HEIGHT; mapRow++)
  {
    for (mapIndex = 0; mapIndex < GAME_PORT_WIDTH; mapIndex++)
    {
      tileIndex = TerrainMap[mapRow][mapIndex];
      if (tileIndex > 0)
      {
        // retrieve tile location in lpDDSOffscreen
       srcRect.left = cells[tileIndex].posx;
       srcRect.top = cells[tileIndex].posy;
       srcRect.right = srcRect.left + tileSize;
       srcRect.bottom = srcRect.top + tileSize;

       // calculate tile destination on lpDDSBack
       destx = mapIndex * tileSize;   // destination x
       desty = mapRow * tileSize;     // destination y

       // blit the tile to the location
       BlitSurface( lpDDSOffscreen, lpDDSBack,
            destx,desty,&srcRect, DDBLTFAST_SRCCOLORKEY);
      }
    } // for mapIndex
  } // for mapRow
}
```

Our code walks the 15-by-20 cell *TerrainMap* array, cell by cell. You retrieve the tile type with the variable *tileIndex* inside the nested *for* loop. You will render all tiles except those with a tile type of 0. In order to blit the tile to its destination on the flipping DirectDraw surface, you must retrieve where the tile is located in the offscreen DirectDraw surface. The tile lookup table, which is your array of cells, contains this information. The tile locations are moved into a Windows RECT structure, *srcRect*, to use when you call the *Spells of Fury* DDTOOLS function BlitSurface(). To draw the tile to its location on the flipping DirectDraw surface, you must also provide a destination. The destination coordinate is where you'll start drawing the upper-left-hand corner of the tile. You use the variables *mapRow* and *mapIndex* to provide the correlation back to the flipping DirectDraw surface coordinates from the *TerrainMap* cells. The code for this operation is located in Listing 9-5.

LISTING 9-5 Drawing tiles to the back flipping surface

```
// calculate tile destination on lpDDSBack
 destx = mapIndex * tileSize;       // destination x
 desty = mapRow * tileSize;         // destination y

 // blit the tile to the location
 BlitSurface( lpDDSOffscreen, lpDDSBack,
        destx,desty,&srcRect, DDBLTFAST_SRCCOLORKEY);
```

The modified version of DITEST's function updateFrame() to update the display is shown in Listing 9-6.

LISTING 9-6 The new updateFrame() function

```
void updateFrame()
{
        checkKeyboard();
        getJoystick();
        EraseBackSurface();
        drawBackGround();
        drawTerrain();
        drawBubonic();
        flipSurface();
}
```

With this code, and your previous code from the DITEST program in Chapter 4, you have provided a more interesting terrain and background. Figure 9-2 shows your new screen image when the TILES program executes. There is one problem with your code though. If you haven't executed the TILES program go ahead and do so. You'll notice that even though the terrain has small hills and valleys, Bubonic the WartHog just seems to walk on thin air over certain tiles.

Non-Linear Terrains

The previous example program, TILES, shows how you could use a simple terrain map to make your DITEST program from Chapter 4 look more interesting. However, even though the terrain appeared flat in your *TerrainMap* array, the actual tiles were non-linear with small hills and valleys in them. From Bubonic's point of view, he was still running on his old flat surface. You need to modify how you draw Bubonic the WartHog for a more realistic effect of Bubonic running over rolling hills. In this discussion, you'll modify the TILES program from the previous example in a new program you'll call SLOPE.

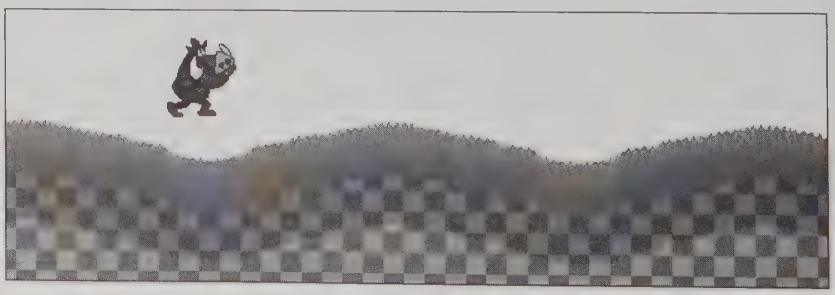


FIGURE 9-2
The TILES program

Using the Slope Equation

You can produce the illusion of Bubonic running through the hills and valleys of your tiles by using the slope equation, $m = dy/dx$, from Chapter 6. Rather than have Bubonic run on thin air, you can use SLOPE to pull your character into the tile, depicting movement on an uneven terrain, as shown in Figure 9-3. To do this you must provide more information about each tile. This means you're going to have to modify your CELL structure used for the tile lookup table in the previous TILES example program. Your new CELL structure is shown in Listing 9-7.

LISTING 9-7 The CELL structure for using slope

```
typedef struct {
    int         posx;       // X position of cell
    int         posy;       // Y position of cell
    int         level;      // Drawing level of cell
    int         frames;     // Number of frames to make a sequence.
    unsigned    delay;      // Time of delay for this cell.
    int         beginLevel; // Beginning base level of cell.
    int         endLevel;   // Ending base level of cell.
} CELL;
```

You now have a more sophisticated CELL structure from the one introduced earlier. Rather than just storing the *x* and *y* coordinate of the upper-left-corner of the tile's location in the offscreen DirectDraw surface, you also include information about tile placement, number of frames in an animation sequence, delay to remain on the screen, and the beginning and ending *y* coordinates of the tile. These last two fields in your structure, declared as *beginLevel* and *endLevel*, are where you'll store values to determine the slope on each tile.

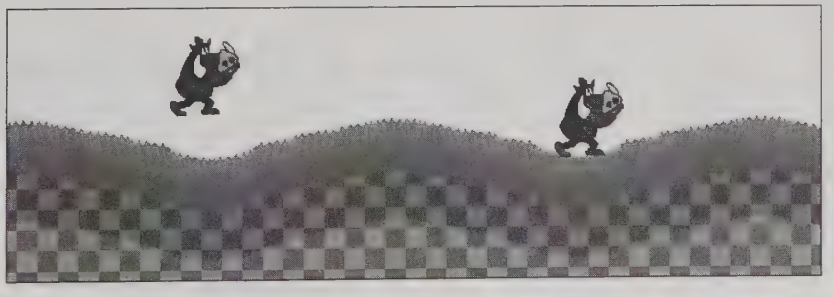


FIGURE 9-3
Using SLOPE to pull Bubonic into tile on uneven terrain

beginLevel and *endLevel* Values

In Chapter 6, you used the value *dy* to determine the change in *y* over a range *dx*. You called these values rise over run. You calculated *dy* as the difference in two *y* values on the same line. In the SLOPE example, you calculate *dy* from the difference of *beginLevel* minus *endLevel*. The values *beginLevel* and *endLevel* are *y* values on each side of the 32-pixel tile. The *beginLevel* value is located at (0, *y*) on the left side of the tile, and *endLevel* at (32, *y*) on the right side of the tile.

The *y* values of *beginLevel* and *endLevel* are produced by examining the tile art. For instance, the tile type 1 is a flat tile; however, the first eight rows of pixels in this 32 x 32 tile are filled with a pixel you'll use as a DirectDraw DDCOLORKEY value for the DirectDraw surface member BltFast().You will want to pull Bubonic down these eight pixels for the blit operation. If you don't pull Bubonic down to the tile image, Bubonic will look like he's doing the Barney Rubbel run in mid-air. So you set the *beginLevel* value to 8 (TILESIZE/4 = 32/4 = 8). The tile depicts a flat surface for Bubonic to run on, so you want a slope of 0. So, you set the *endLevel* value to TILESIZE/4 as well. This is a simple case where Bubonic is being pulled closer into a tile, but the terrain is flat, as shown in Figure 9-4.

You see from Figure 9-4 that tile type 11 has slope. By eyeballing the tile using the ZOOM function in the Microsoft Paint accessory, you provide a slope using the values *beginLevel* equals TILESIZE/8 and *endLevel* equals TILESIZE/4. The *beginLevel* of tile 11 doesn't exactly line up to *endLevel* of tile 6, but these values do provide a nice steady slope with the other tiles in the sequence of base tiles. After painstakingly building your tile database, your new cells array is represented in Listing 9-8.

LISTING 9-8 Tile cells lookup table

```
CELL    cells[] = {
//X     Y       LEVEL       F   D   BL           EL           Description
0,      0,      0,          0,  0,  NOLEVEL,     NOLEVEL,
1,      482,    INBACK,     0,  0,  TILESIZE/4,  TILESIZE/4,
1,      516,    INBACK,     0,  0,  0,           0,
1,      550,    INBACK,     0,  0,  0,           0,
1,      584,    INBACK,     0,  0,  0,           0,
1,      618,    INBACK,     0,  0,  TILESIZE/2,  TILESIZE/2,
35,     482,    INBACK,     0,  0,  TILESIZE/4,  TILESIZE/4,
            :
273,    550,    INBACK,     0,  0,  0,           0,
307,    482,    INBACK,     0,  0,  TILESIZE/4,  TILESIZE/8,
307,    516,    INBACK,     0,  0,  0,           0,
307,    550,    INBACK,     0,  0,  0,           0,
};
```

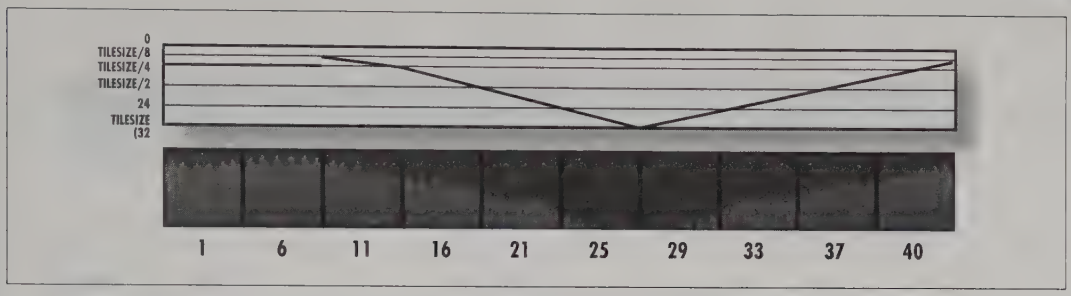


FIGURE 9-4 Base tiles with *beginLevel* and *endLevel* slopes

Note the last two entries in each row are the *beginLevel* and *endLevel* field values. Figure 9-4 shows the tiles and their graphical slope representations of tile types 1, 6, 11, 16, 21, 25, and 29. These are the base tiles that Bubonic will walk, run, or stand on. If you are wondering about the defined values INBACK and INFRONT in the cells array, these are flags to tell your code if this tile is to be drawn in front of or behind the Bubonic character on the DirectDraw surface.



The Slope Lookup Table

The *beginLevel* and *endLevel* values are stored in the cells table and they are indexed by the cell type. For convenience, you are going to build a simple slope table that will be indexed by a logical screen coordinate. Remember you divided the 640-pixel-wide screen by 32 for a logical screen width of 20. Your array integers, which you call *slopeTable*, will contain 20 entries. Each entry in *slopeTable* will contain the *delta y* value corresponding to the base tile located at that logical screen coordinate, as shown in Figure 9-5. The *delta y* value is the difference of the *beginLevel* value minus the *endLevel* value. Your code to build the *slopeTable* is shown in Listing 9-9.

LISTING 9-9 Constructing the slope table for the example

```
int slopeTable[GAME_PORT_WIDTH];
void buildSlopeTable()
{
  int tileIndex, mapIndex;
  int dy;

  for (mapIndex = 0; mapIndex < GAME_PORT_WIDTH; mapIndex++)
  {
    tileIndex = TerrainMap[BASE_ROW][mapIndex];
    dy = cells[tileIndex].beginLevel - cells[tileIndex].endLevel;
    slopeTable[mapIndex] = dy;
  }
}
```

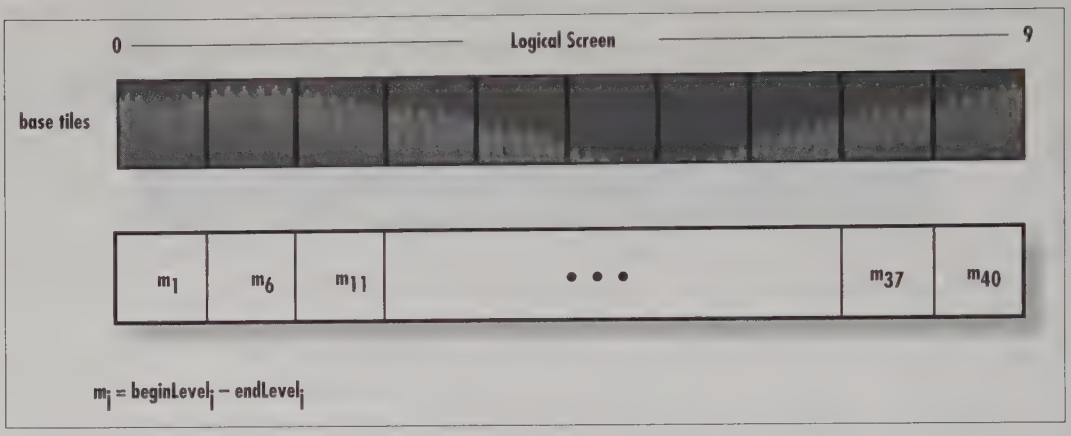


FIGURE 9-5 Slope table entries

All the base tiles that Bubonic will walk on in this example are stored in row 11, defined as the BASE_ROW, of the *TerrainMap*. The *slopeTable* array is constructed by walking the *TerrainMap* with the row index held constant at BASE_ROW. Using the *for* loop, you walk each cell in the BASE_ROW of the *TerrainMap* and retrieve the tile type of the base tile stored in that cell. The tile type is the tile index (tileIndex in code) in the cell's database. The desired *beginLevel* and *endLevel* values can be retrieved for calculating the *delta y* value at that logical screen location. The *delta y* value is stored in the *slopeTable* array for quick lookups.



Drawing Bubonic Using Slope

Finally you reach your objective: to draw Bubonic so it appears he's running over his bumpy terrain. To accomplish this you have included *y* values in your cells database to compute *delta y* values for the base tiles. The *delta y* for each base tile is stored in the *slopeTable* array based on its logical screen position. You'll modify the function drawBubonic() from the DITEST and TILES example to fit your needs. The code for drawBubonic() is provided in Listing 9-10.

LISTING 9-10 The drawBubonic function

```
void drawBubonic()
{
  DWORD thisTickCount;
  DWORD tickDiff;
  int   mapIndex;
  int   base;
```

```
  thisTickCount = GetTickCount();
  tickDiff = thisTickCount - wartHog.lastTickCount;

  mapIndex = (wartHog.posx - 32) / TILESIZE;
  base = 10 * TILESIZE;
  wartHog.posy = base  - slopeTable[mapIndex] - TILESIZE/4;
  switch(wartHog.direction)
  {
    case HOG_STOP_RIGHT:
      drawOneToBack(wartHog.posx,
      wartHog.posy, standRightRect[wartHog.currentCell], FALSE);
      if (tickDiff > wartHog.delay)
      {
        if (wartHog.currentCell++ > MAX_STOP_CELLS-1)
          wartHog.currentCell = 0;
        wartHog.lastTickCount = GetTickCount();
      }
      return;
    case HOG_STOP_LEFT:
      drawOneToBack(wartHog.posx,
      wartHog.posy, standLeftRect[wartHog.currentCell], FALSE);
      if (tickDiff > wartHog.delay)
      {
        if (wartHog.currentCell++ > MAX_STOP_CELLS-1)
          wartHog.currentCell = 0;
        wartHog.lastTickCount = GetTickCount();
      }
            return;
    case HOG_GO_RIGHT:
      drawOneToBack(wartHog.posx,
      wartHog.posy, goRightRect[wartHog.currentCell], FALSE);
      if (tickDiff > wartHog.delay)
      {
        if (wartHog.currentCell++ > MAX_WALK_CELLS-1)
          wartHog.currentCell = 0;
        wartHog.lastTickCount = GetTickCount();
      }
      return;
    case HOG_GO_LEFT:
      drawOneToBack(wartHog.posx,
      wartHog.posy, goLeftRect[wartHog.currentCell], FALSE);
      if (tickDiff > wartHog.delay)
      {
        if (wartHog.currentCell++ > MAX_WALK_CELLS-1)
          wartHog.currentCell = 0;
        wartHog.lastTickCount = GetTickCount();
      }
      return;
    default:
      break;
  } // end switch
}
```

Using the *slopeTable*, you only added five lines to the entire function. All of this code was added at the beginning of drawBubonic().You included a variable for the *mapIndex*, which is the index to the logical screen coordinate. You included a base variable to compute the upper-left-hand destination corner of the Bubonic sprite, as shown in Listing 9-11.

LISTING 9-11 Locating the base tile for Bubonic to stand on

```
int mapIndex;
int base;

mapIndex = (wartHog.posx - 32) / TILESIZE;
base = 10 * TILESIZE;
wartHog.posy = base  - slopeTable[mapIndex] - TILESIZE/4;
```

The calculation for the *mapIndex* is based on the position of Bubonic on the physical DirectDraw surface. Originally, I computed the *mapIndex* as wartHog.posx/TILESIZE. But sometimes this left me with strange results, because Bubonic is 64 x 64 pixels in size and the tiles are 32 x 32. The tiles are half his size, so if the tile suddenly went down quickly, Bubonic had a delayed effect of going down. By retrieving the *mapIndex* relative to the center of the sprite, *wartHog.posx* - 32 produced better results. So the *mapIndex* value is a mapping from the physical DirectDraw surface coordinate *wartHog.posx* to the logical screen coordinate *mapIndex*.

The value of the base is where the top of Bubonic's destination RECT would be if his terrain were perfectly flat. Remember, the base tiles are all in row 11 of the terrain map. If the tiles were perfect 32 x 32 pixel square tiles, then Bubonic would run on row 10 of the terrain map. Well, your terrain isn't so perfect. So this is your base value to work with. You start out by setting Bubonic's base value to 10 * TILESIZE. That is, if you were to actually draw Bubonic, *wartHog.posy* would be 320. Using your base value you need to use the *delta y* values from the slope table to pull Bubonic into the tiles as desired. I added an additional offset of TILESIZE/4 because I found it provided a good finishing touch.



Scrolling Terrain Example

The discussion of tiles in the previous section was limited to fixed screens. You used a 20 x 15 logical screen to represent your 640 x 480 flipping DirectDraw surfaces. An array of 20 x 15 cells, called a terrain map, was used to identify where you wanted tiles placed in the logical screen. The intention was that the Bubonic character would never leave the confines of the 20 x 15 logical screen boundaries. This made a really interesting demo but it really didn't model the behavior of a real video game, such as *Sonic the Hedgehog* or *Super Mario Brothers*.

In these console-style games, the screen scrolls right or left as the character moves. In this section, I'll discuss how to build a terrain larger than the 20 x 15 logical screen coordinates. You'll see how to use the logical screen as a window into your virtual terrain.



Column Slices

In your static tile example program, TILES, you mapped the tiles by row. The logical screen and the terrain map were the same size arrays with 20 x 15 cells each. With a scrolling terrain, you're going to change things a bit. In the static tile example, you populated only the last four rows of the *TerrainMap* array with tile types. This meant you had 11 rows of dead space in your 20 x 15 map array. Your new logical terrain map will be *n* columns long. You can minimize the number of dead space rows by shrinking your terrain map down from 15 rows to 10, as shown in Figure 9-6. Using this configuration, you won't draw any terrain tiles in the first five rows of the logical screen. You'll assume this space is to be reserved for the sky or a backdrop.

Each column in the new terrain map is a vertical *slice* of the terrain. One slice in the terrain map is one tile (32 pixels) wide and ten tiles tall. The maximum number of slices stored in your terrain map is represented by the value numSlices. If there are 48 slices in the entire map, then *numSlices* equals 48.

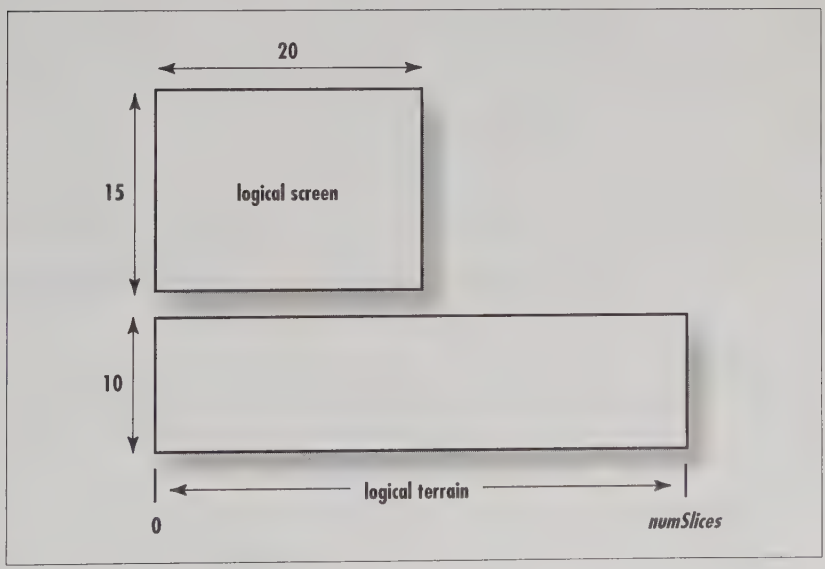


FIGURE 9-6
The logical screen and terrain architecture

Slicing and Dicing the Logical Terrain

The logical terrain is larger than the *TerrainMap* array presented in the last two examples, TILES and SLOPE. Some game developers write a map editor for their project so an artist or another game designer can lay out the map. When the map is completed, it is stored in a file where it can be retrieved by the game application. You will store your map in a text file called TERRAIN.INF. Your terrain map will be stored vertically in the file, as opposed to horizontally, as you did with the *TerrainMap* array from the previous example. Each text row in the file is a column slice as shown in Figure 9-7. A slice is 30 bytes long and contains 10 entries per slice. Each entry is a value 0 - 99 and is separated by a white space, making each entry 3 bytes in length. The total byte size of the slice for 10 entries per slice is therefore 30 bytes.

The TERRAIN.INF map can be read in using the four steps shown in Figure 9-8. The process requires you to open the TERRAIN.INF file, get the size of file, allocate the slice data structure to hold the file contents, and then read the data in.

Step 1. Opening the TERRAIN.INF File

You can read in the TERRAIN.INF slices using standard C library function calls. You're going to open the file, get the file size (*fileSize*) to determine how many slices are stored in the file, compute the *numSlices* value, and then allocate memory for the terrain slice table. The code to open the file and allocate slice table memory is shown in Listing 9-12.

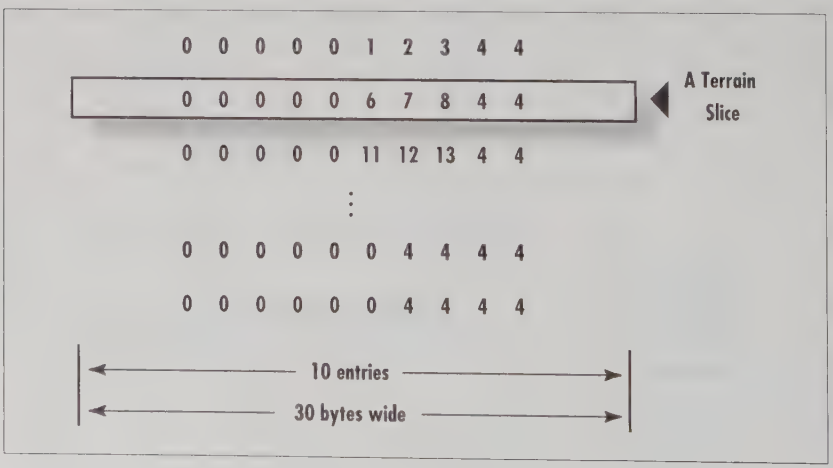


FIGURE 9-7
TERRAIN.INF file format

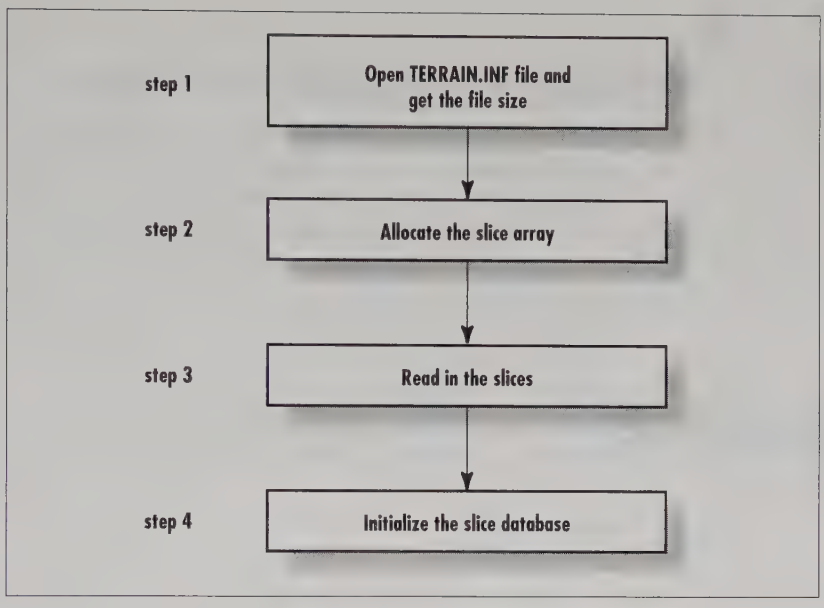


FIGURE 9-8
The four steps to read in the TERRAIN.INF file

LISTING 9-12 Reading the *Bubonic* terrain map from a file

```
if(!(stream = fopen( "TERRAIN.INF", "r")))
{
  MessageBox(HWND_DESKTOP,"Error Reading TERRAIN.INF",
                                    "Spells of Fury", MB_OK);
  return FALSE;
}

fseek(stream, 0L, SEEK_END);
fgetpos(stream, &fileSize );
fseek(stream, 0L, SEEK_SET);
numSlices = (int) (fileSize / 30);

slicePtr = (SLICE *)LocalAlloc( LPTR, numSlices * sizeof(SLICE) );
if (slicePtr == NULL)
  return FALSE;
```

You use the C library fseek() and fgetpos() function calls to determine the byte size of your file. Since each slice entry is 30 bytes in length, you can divide the *fileSize* value by 30 to determine how many terrain slices are in the map file. This will be your numSlices value, which you'll need to allocate memory for the slices.



Step 2. Allocating the SLICE Data Structure

Before you allocate memory for the slices, let's discuss the SLICE structure that will be used to maintain the terrain tile SLICE database. The SLICE database contains information about your terrain tile map stored in columns in the TERRAIN.INF file. The fields in the SLICE database include an array of the tile types in that slice of the map. The SLICE fields also include the *base* tile *beginLevel* and *endLevel* for each slice in the map. Remember, you used these values in the last discussion to compute slope for Bubonic the WartHog. The SLICE data structure is shown in Listing 9-13.

LISTING 9-13 The SLICE map data structures

```
typedef struct {
    int     type;
    int     frameNum;
    DWORD   lastTickCount;
} CONTENT;

typedef struct {
    CONTENT contents[NUMCONTENTS];  // Cell types in this slice
    int     beginLevel;             // Base height at beginning
    int     endLevel;               // Base height at end of slice.
    float   slope;                  // Slope of base.
    char    trap;                   // Is slice a trap?
} SLICE;
```

The CONTENT field of the SLICE database is an array of tile types in that slice. There are ten tiles per slice in the map file; therefore, ten entries per SLICE in the CONTENT array field. You don't know how many SLICE entries there will be in the database until you open the file and calculate the numSlices value as you did in Step 1. Using the *numSlices* value, you allocate memory for the SLICE database with the following code:

```
  slicePtr = (SLICE *)LocalAlloc( LPTR, numSlices * sizeof(SLICE) );
if (slicePtr == NULL)
  return FALSE;
```

Once the SLICE database is allocated, you are ready to start reading in the column entries from the map file. The key point from Step 2 is that the SLICE database stores information about the terrain map in columns, rather than rows, as you did with the previous examples. The *slopeTable* array is also eliminated because the *beginLevel* and *endLevel* values for the base tile in each slice are stored as an entry in the SLICE database.

Step 3. Reading the Tiles into the SLICE Database

Each 30-byte row in your TERRAIN.INF is a column entry for SLICE database. From Step 1, you determined how many entries, *numSlices*, were in the file. You'll now read the terrain tile information, by column slice, into the database. This requires nested *for* loops to initialize the database with the file entries. The outer loop iterates *numSlices* times to read in each row of data from the file. An inner loop reads in the tile data for each slice. The code for this is provided in Listing 9-14.

LISTING 9-14 Reading the SLICE data from the TERRAIN.INF file

```
#define NUMCONTENTS 10
for (i = 0; i < numSlices; i++)
{
  for(j = 0; j < NUMCONTENTS; j++)
  {
    if (j == NUMCONTENTS-1)
      fscanf(stream,"%d\n",&slicePtr[i].contents[j].type);
    else
      fscanf(stream,"%d",&slicePtr[i].contents[j].type);
  }
}
fclose(stream);
```

Use the C library routine fscanf() to read in formatted data from the file. There are ten (NUMCONTENTS) entries per row in the file. Each entry is 3 bytes in length. This magic number allows the tile types to hold a range of values from 0 to 99, and each to be separated by a white space. The C run-time library function fscanf() reads in formatted input from the file and ignores the blank spaces (' '), tabs (\t) and newlines (\n). The newline control code is physically present in your TERRAIN.INF, so you compensate for this using the following code:

```
if (j == NUMCONTENTS-1)
       scanf(stream,"%d\n",&slicePtr[i].contents[j].type);
else
       scanf(stream,"%d",&slicePtr[i].contents[j].type);
```

The fscanf() reads past the white spaces because you specified the formatted input as "%d". However, the entry at NUMCONTENTS-1, the last entry in the row of the file, has the newline code. You must specify a different format, "%d\n", to read the byte data in as an integer. When the outer loop for *numSlices* terminates, all of your tile types will be stored columnwise in the SLICE database, as shown in Figure 9-9.

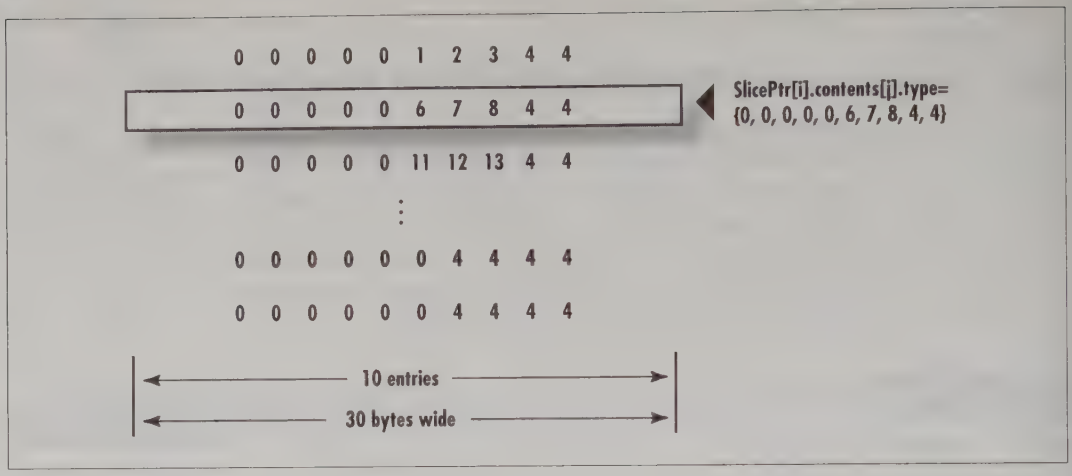


FIGURE 9-9 Column of tile entries in the SLICE database



Step 4. Initializing the Base Tile Information

In this step you initialize the *beginLevel* and *endLevel* values for the base tile in this slice. A *base* tile is the tile Bubonic will walk when you animate this slice. When you demonstrated the animation of Bubonic walking in your previous example, you used a *slopeTable* to tell how you would draw the character. You needed this because you placed all your base tiles in the *TerrainMap* in row 11. All values in the *TerrainMap* at row 11 were non-zero. Here you won't use a base row. You place the base tile anywhere you want in the slice. This means a base tile can be in the database at *slicePtr[2].contents[3].type* or it can be at *slicePtr[2].contents.[9].type*. This means your terrain can really be nonuniform with cliffs, gorges, and so on. How do you identify the base tile in the slice database? In your definition of a base tile, it is the first non-zero tile type in each slice. Remember, a tile type of 0 is a dummy place-holder for no tile to be drawn. The first non-zero value in the slice is your base tile. Take a look at Figure 9-9 where you show your pseudo array of contents for the second slice in the file. The array *contents[].type* has the values {0, 0, 0, 0, 6, 7, 8, 4, 4}. The tile type 6 will be your base tile. If you look up the tile types from Figure 9-1, you'll see that tile type 6 is grass. One way to think of it is that you're laying virtual sod. The code to initialize the SLICE database with base tile information is shown in Listing 9-15.

LISTING 9-15 Initializing the base tile information

```
#define MAXCONTENTS 15
for (i = 0; i < numSlices; i++)
{
  j = 0;
  found = FALSE;
  while(!found)
  {
    if ( (slicePtr[i].contents[j].type != 0)
      && (cells[slicePtr[i].contents[j].type].beginLevel != NOLEVEL))
      found = TRUE;
    else
      j++;
  }
  base = j * CELLSIZE + ((MAXCONTENTS-NUMCONTENTS)*CELLSIZE);
  slicePtr[i].beginLevel = base
                    + cells[slicePtr[i].contents[j].type].beginLevel;
  slicePtr[i].endLevel = base
                    + cells[slicePtr[i].contents[j].type].endLevel;
  slicePtr[i].slope = (float)(slicePtr[i].beginLevel
                    - slicePtr[i].endLevel)/CELLSIZE;
  slicePtr[i].trap = FALSE;
  if (slicePtr[i].contents[j].type == 43)
    slicePtr[i].trap = TRUE;
}
```

The *while(!found)* loop is what you use to find the base tile in this slice of your map. Here you are iterating the *contents[].type* field of your database to find the first non-zero tile type. When the base tile is located you are ready to calculate the slope for the values. The slope values in the TERRAIN programming example are physical screen coordinate values. The value base in this example is where you would blit Bubonic if the tile your character is on is perfectly square, such as tile 8 in Figure 9-1. You calculate the base using the following code:

```
base = j * CELLSIZE + ((MAXCONTENTS-NUMCONTENTS)*CELLSIZE);
```

Using your example of tile type 6, which, coincidently, is the sixth entry in *slicePtr[2].contents[j=5].type*, it is physically located at (j=5 * CELLSIZE). CELLSIZE is the height of your tile, 32 pixels. This would be the physical screen *y* coordinate value except your logical screen and terrain are different sizes. Remember, the logical screen is 20 x 15 and the terrain *numSlices* x 10. You need to add an offset to correct for the terrain map size difference. This is your computation, (MAXCONTENTS-NUMCONTENTS)*CELLSIZE, which provides the correct mapping to the physical DirectDraw surface *y*-coordinate.

The SLICE database contains the *beginLevel* and *endLevel* values for the base tile in the slice. These values are retrieved from the cells array. You know from previous discussions that the tile type is an index into the cells array database. Retrieving the *beginLevel* and *endLevel* values, you add these to the base *y* coordinate value. This value is the physical DirectDraw surface *beginLevel* and *endLevel* *y* coordinates for the base tile in the slice. You also compute the slope for the base tile in the SLICE database, which looks like this:

```
slicePtr[i].beginLevel = base
                  + cells[slicePtr[i].contents[j].type].beginLevel;
slicePtr[i].endLevel = base + cells[slicePtr[i].contents[j].type].endLevel;
slicePtr[i].slope = (float)(slicePtr[i].beginLevel
                  - slicePtr[i].endLevel)/CELLSIZE;
```

The *beginLevel* and *endLevel* values for tile type 6 are shown in Figure 9-10. Here you see a tile slice at DirectDraw surface coordinate *x*. The base tile for tile type 6 is positioned at (*x*,320).

The example programs TERRAIN and BUBONIC use the function initSlices() to perform all three steps in initializing the SLICE database. The code for initSlices() is provided in Listing 9-16.

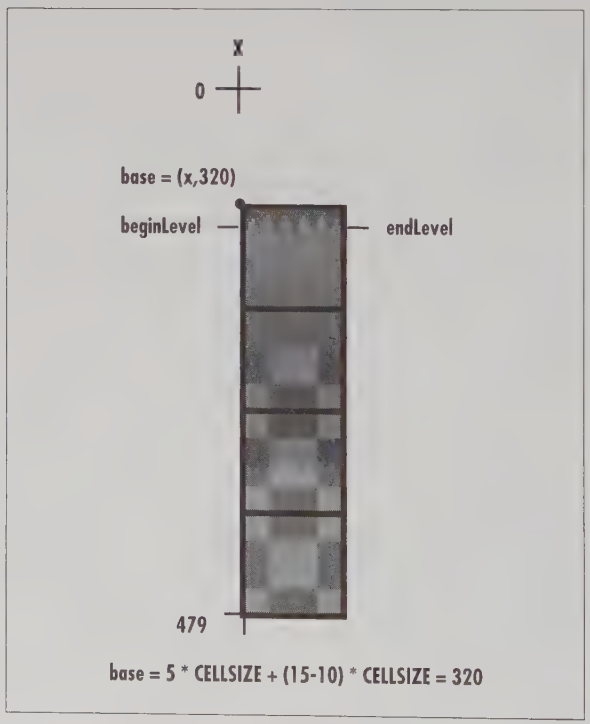


FIGURE 9-10
The base tile data stored in the SLICE database

LISTING 9-16 The initSlices() function used in the TERRAIN example program

```
BOOL initSlices( void )
{
    FILE     *stream;
    int      i,j;
    int      base;
    fpos_t   fileSize;
    BOOL     found;

    // open the terrain.inf file for reading.
    if(!(stream = fopen( "TERRAIN.INF", "r")))
    {
        MessageBox(GetActiveWindow(),
       "Error Reading TERRAIN.INF", "Bubonic", MB_OK);
        return FALSE;
    }

    // get the file size
    fseek(stream,0L,SEEK_END);
    fgetpos( stream, &fileSize );
    fseek(stream,0L,SEEK_SET);
    numSlices = (int)fileSize / 30;

    // allocate the SLICE database memory
    slicePtr = (SLICE *)LocalAlloc(LPTR, numSlices * sizeof(SLICE));
    if (slicePtr == NULL)
      return FALSE;

    // Go through the file and initialize the terrain array.
    for (i = 0; i < numSlices; i++)
      for(j = 0; j < NUMCONTENTS; j++)
      {
        if (j == NUMCONTENTS-1)
          fscanf(stream,"%d\n",&slicePtr[i].contents[j].type);
        else
          fscanf(stream,"%d",&slicePtr[i].contents[j].type);
      }
    fclose(stream);

    // initialize the base tile information in each slice
    for (i = 0; i < numSlices; i++)
    {
       j = 0;
       found = FALSE;
       while(!found)
       {
```

continued on next page

continued from previous page

```
            if ((slicePtr[i].contents[j].type != 0)
               && (cells[slicePtr[i].contents[j].type].beginLevel
               != NOLEVEL))
              found = TRUE;
            else
              j++;
          }

          base = j * CELLSIZE + ((MAXCONTENTS-NUMCONTENTS)*CELLSIZE);
          slicePtr[i].beginLevel = base
                       + cells[slicePtr[i].contents[j].type].beginLevel;
          slicePtr[i].endLevel = base
                       + cells[slicePtr[i].contents[j].type].endLevel;
          slicePtr[i].slope = (float)(slicePtr[i].beginLevel
                       - slicePtr[i].endLevel)/CELLSIZE;
          slicePtr[i].trap = FALSE;
          if (slicePtr[i].contents[j].type == 43)
            slicePtr[i].trap = TRUE;
        } // for numSlices
        return TRUE;
}
```

The SLICE database contains your virtual terrain. Now you need to provide some code to render your landscape. The next topic will examine how you'll go about displaying your tiles.



Drawing the Terrain Slices

One way you could draw all the slices is to allocate an offscreen DirectDraw surface large enough to hold all of your tile slices. Assuming there are 48 slices in the TERRAIN.INF file (numSlices = 48), you will need a DirectDraw surface that is numSlices x CELLSIZE (48 x 32) or 1536 pixels wide by 480 pixels tall. You could then use the DirectDraw member BltFast() and slide a 640 x 480 window over this offscreen DirectDraw surface. Or, you could reserve your precious video memory for sprites and tiles, and render your virtual terrain using logical coordinates. This alternative doesn't take up any more memory than it takes to store the tile information in the SLICE database. A logical mapping is what you're going to use to draw the terrain tiles from the SLICE database. To do this, you are going to use an index, *screenPosx*, that maps the logical map to the flipping DirectDraw surface lpDDSBack. The variable *screenPosx* slides back and forth on your logical map. The *screenPosx* index has a range of 0 to (numSlices*CELLSIZE) - 640. The *screenPosx* value represents the start of the terrain to be rendered to *lpDDSBack* at its location x=0, as shown in Figure 9-11.

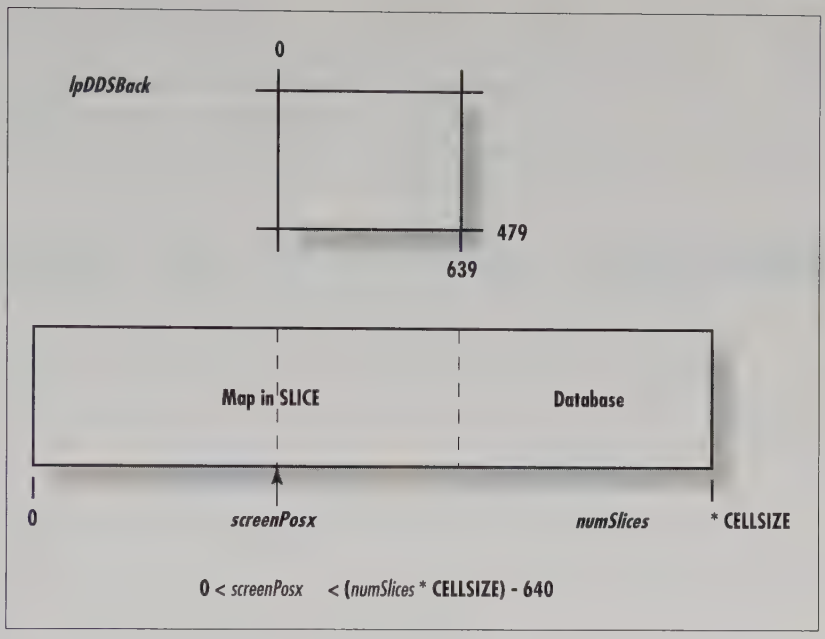


FIGURE 9-11
Using *screenPosx* to index the logical terrain in the SLICE database

What you're doing is creating a sliding window into the SLICE database contents fields. The start of the sliding window is at *screenPosx*. The width of *screenPosx* is 640 pixels. As you have noticed, *screenPosx* is in screen coordinates and the tiles in the SLICE database are logical coordinates. The value *screenPosx* is determined by the SLICE database index times the CELLSIZE. In the TERRAIN and BUBONIC example programs, you update the *screenPosx* value and map it back into a slice index. The slice index is simply the *screenPosx* value divided by the CELLSIZE. The procedure to draw the tile cells requires two steps. In Step 1, you check for fractions of tiles, and in step 2, you draw the map slice to the DirectDraw surface, *lpDDSBack*.



Step 1. Drawing the First Slice in a Sliding Window

In the TERRAIN example program, the user presses the VK_4 or VK_6 key to go left or right. When this event occurs, you update the *screenPosx* value by -8 pixels to go left or +8 pixels to go right. You're not moving in a manner that is uniform (the tiles are 32 pixels wide) to your tile mapping in the SLICE database. When you perform the blit, you may actually be overlapping a fraction of tile.

The function drawCells(), used by the TERRAIN and BUBONIC programs, is what you'll use to render the tiles from the SLICE database onto the

DirectDraw surface. The function is essentially two nested loops: the outer loop iterates the number of slices, *sliceNumber*, that you want to draw to the DirectDraw surface. The DirectDraw surface is 640 pixels wide, which, divided by CELLSIZE, gives you a *sliceNumber* of 20. That is, you can draw 20 slices to the DirectDraw surface. Your outer loop will make 20 iterations. The code for the slice iteration loop is shown in Listing 9-17.

LISTING 9-17 Looping through the terrain map slices

```
int     lengthLeft, portionSize, portionOffset;
int     currentPosX, currentPosY;
int     i,j;
RECT    rcRect;

lengthLeft = SCREENSIZEX;

for (i = 0; i < sliceNumber; i++)
{
  // If first slice then most likely this will be a portion
  // of the slice.
  if (i == 0)
  {
    portionSize = CELLSIZE - (screenPosx % CELLSIZE);
    portionOffset = (screenPosx % CELLSIZE);
  }
  else
  {
    portionOffset = 0;
    if (lengthLeft > CELLSIZE)
      portionSize = CELLSIZE;
    else
      portionSize = lengthLeft;
  }   // if
  // draw map slices here
}
```

When you make the first iteration, examine the *screenPosx* value to determine if you have a case where you are drawing a fractional tile from a previous slice. The variable *portionSize* tells you how much of the previous slice is still visible during this blit. You must also determine if you need to offset the tile's destination coordinates. The value *portionOffset* tells you by how much you must offset the tiles in the *x* direction when you call the blitting routine. The *portionOffset* is only calculated when you draw the first slice. Once this value is determined, you enter your code where you start rendering the tiles.



Step 2. Drawing Slices to the DirectDrawSurface

The inner loop in your drawCells() function walks the contents array of each slice in the database. If the tile type is non-zero and the desired *Level* is set, you

extract the tile data from the SLICE database. Remember, the *Level* is a value stored in the *cells* database. This tells you if the tile is to be drawn in front of (INFRONT) or behind (INBACK) the Bubonic character. In this example, you want the tiles that are to be drawn behind Bubonic, so the tile level is set to INBACK. If you have met this criteria, then the code for your inner loop looks like this Listing 9-18.

LISTING 9-18 Determining whether to draw a tile in front or back

```
rcRect.left = cells[slicePtr[firstSlice+i].contents[j].type].posx +
portionOffset;
// If cell is part of a sequence then find if next frame needs to be
// shown.
if (cells[slicePtr[firstSlice+i].contents[j].type].frames != 0)
{
  if((thisTickCount
        - slicePtr[firstSlice+i].contents[j].lastTickCount)
         > (cells[slicePtr[firstSlice+i].contents[j].type].delay))
  {
    slicePtr[firstSlice+i].contents[j].lastTickCount = thisTickCount;
    slicePtr[firstSlice+i].contents[j].frameNum++;
    slicePtr[firstSlice+i].contents[j].frameNum %=
    cells[slicePtr[firstSlice+i].contents[j].type].frames;
  }
  rcRect.left += slicePtr[firstSlice+i].contents[j].frameNum * BCELLSIZE;
}
rcRect.top = cells[slicePtr[firstSlice+i].contents[j].type].posy;
rcRect.right = rcRect.left + portionSize;
rcRect.bottom = rcRect.top + CELLSIZE;
currentPosY = (j * CELLSIZE) + ((MAXCONTENTS-NUMCONTENTS)*CELLSIZE);
drawOneToBack(currentPosX, currentPosY , rcRect, FALSE);
```

The variable *firstSlice* is the first slice you are going to draw to the flipping DirectDraw surface, *lpDDSBack*. The value of *firstSlice* is determined by the *screenPosx* value divided by the CELLSIZE (tile size = 32). The *firstSlice* variable is your coordinate mapping from DirectDraw surface style coordinates back to logical slice map coordinates, as shown in Figure 9-11.



The Tile Lookup Process

You are starting to retrieve the tile location as it is stored on the offscreen DirectDraw surface, *lpDDSOne*. This code may look a little confusing because you are accessing the tile cells database based on information stored in the SLICE database. The code for the left RECT coordinate of the tile you are looking up looks as follows:

```
rcRect.left = cells[slicePtr[firstSlice+i].contents[j].type].posx +
➥portionOffset;
```

The Windows RECT structure, rcRect, is being set to the physical location of the tile in *lpDDSOne*. The location of the tile is stored in the cells database, in fields *posx* and *posy*. This coordinate pair is the upper left corner of the tile as it is located in *lpDDSOne*. The specific tile type you want to draw is located in the SLICE database, in the SLICE contents array. By extracting the tile type, you can index the desired tile in the cells database. Remember, the tile type is the index into the cells array. Once you locate the tile's left coordinate, you add the *portionOffset*. This will properly shift you into the tile if you're not drawing perfect 32 x 32 pixel tiles.

Once you have located the left coordinate of the tile, you can easily look up the top coordinate as well. The code to determine the tile source RECT for blitting looks like this:

```
rcRect.top = cells[slicePtr[firstSlice+i].contents[j].type].posy;
rcRect.right = rcRect.left + portionSize;
rcRect.bottom = rcRect.top + CELLSIZE;
```



Tiles with Animation Sequences

The tiles discussed so far don't have more than one cell of animation. You'll now examine tiles that do have animation sequences in your terrain. The TERRAIN example program has tiles with spikes in the terrain map. You'll use the SLICE database and your drawCells() function to tell which spike tile animation frame to draw. In the drawCells() function, use the tile type from the SLICE database to provide an index into the cells database. The cells database contains the information regarding the number of frames in the tile's cell animation sequence. This code, which determines the frames of animation, is shown in Listing 9-19.

LISTING 9-19 Number of animation frames and delay for drawing cells

```
if (cells[slicePtr[firstSlice+i].contents[j].type].frames != 0)
{
  if((thisTickCount
        - slicePtr[firstSlice+i].contents[j].lastTickCount)
        > (cells[slicePtr[firstSlice+i].contents[j].type].delay))
  {
    slicePtr[firstSlice+i].contents[j].lastTickCount = thisTickCount;
    slicePtr[firstSlice+i].contents[j].frameNum++;
    slicePtr[firstSlice+i].contents[j].frameNum %=
    cells[slicePtr[firstSlice+i].contents[j].type].frames;
  }
  rcRect.left += slicePtr[firstSlice+i].contents[j].frameNum * BCELLSIZE;
}
```

In the event the tile cell, such as your spikes tile cell, has an animation frame count greater than zero, you must determine which cell to draw and for how long. Remember back in Chapter 4, when you designed the DITEST program? You had to provide an animation frame delay to which one frame would be consistently drawn (multiple DirectDraw surface flips) for so many milliseconds. The delay kept the Bubonic character animation sequence from cycling through all the frames as fast as the computer was capable of drawing them. You're going to do the same for your tile animation. The cells database contains the number of frames per animation sequence and the delay, in milliseconds, for each frame to be consecutively drawn. The SLICE database contains information, in milliseconds, as to when the last tile cell in the frame sequence was drawn. This value is stored in the SLICE database *lastTickCount* field, which is set by the Windows API call GetTickCount(). The code for function drawCells() is provided in Listing 9-20.

LISTING 9-20 Source code for the drawCells()function

```
void drawCells( int sliceNumber, int firstSlice, int Level)
{
  int     lengthLeft, portionSize, portionOffset;
  int     currentPosX, currentPosY;
  int     i,j;
  RECT    rcRect;
  DWORD   thisTickCount;

  lengthLeft = SCREENSIZEX;
  thisTickCount = GetTickCount();

  for (i = 0; i < sliceNumber; i++)
  {
    // If first slice then most likely this will be a portion
    // of the slice.
    if (i == 0)
    {
      portionSize = CELLSIZE - (screenPosx % CELLSIZE);
      portionOffset = (screenPosx % CELLSIZE);
    }
    else
    {
      portionOffset = 0;
      if (lengthLeft > CELLSIZE)
        portionSize = CELLSIZE;
      else
        portionSize = lengthLeft;
    }   // if

    currentPosX = SCREENSIZEX - lengthLeft;
    // Draw the slice of the map onto the screen.
```

continued on next page

continued from previous page

```
    for (j = NUMCONTENTS-1; j > 0; j--)
    {
      if ( (slicePtr[firstSlice+i].contents[j].type != 0)
         && (cells[slicePtr[firstSlice+i].contents[j].type].level
         == Level))
      {
        rcRect.left =
        cells[slicePtr[firstSlice+i].contents[j].type].posx
                              + portionOffset;
        // If cell is part of a sequence then
        // find if next frame needs to be
        // shown.
        if (cells[slicePtr[firstSlice+i].contents[j].type].frames
            != 0)
        {
          if((thisTickCount
            - slicePtr[firstSlice+i].contents[j].lastTickCount)
            > (cells[slicePtr[firstSlice+i].contents[j].type].delay))
          {
            slicePtr[firstSlice+i].contents[j].lastTickCount =
                                                  thisTickCount;
            slicePtr[firstSlice+i].contents[j].frameNum++;
            slicePtr[firstSlice+i].contents[j].frameNum %=
          }
          rcRect.left += slicePtr[firstSlice+i].contents[j].frameNum
                       * BCELLSIZE;
        }
        rcRect.top =
               cells[slicePtr[firstSlice+i].contents[j].type].posy;
        rcRect.right = rcRect.left + portionSize;
        rcRect.bottom = rcRect.top +CELLSIZE;
        currentPosY =
            (j * CELLSIZE) + ((MAXCONTENTS-NUMCONTENTS)*CELLSIZE);
        drawOneToBack(currentPosX, currentPosY , rcRect, FALSE);
      } // if
    } // for
    lengthLeft -= portionSize;
  }   // for
  return;
}
```

You may want to comment out some lines in the code where you determine frame delays to see how the animation behaves without these timers. Turning off the delay on a Pentium system with an S3 or an ATi graphics card may result in a fuzzy ghost image of spikes. This is a result of the faster CPU and graphics card cycling through the cells faster than the human brain is capable of visually processing the image. You could put subliminal messages in your demo for customers to buy your product. Just don't flash images of popcorn or other theater snacks; that's illegal!



The drawCells Driver for the TERRAIN Program

The TERRAIN example program calls the function drawCells from the function updateFrame() just as the BUBONIC program does. The updateFrame() function in the TERRAIN program is specifically designed to be a driver for the drawCells() function. The code for this driver is provided in Listing 9-21.

LISTING 9-21 The drawCells() used for TERRAIN function

```
void updateFrame()
{
    int sliceNumber, firstSlice;

    checkKeyboard();
    getJoystick();
    EraseBackSurface();
    drawBackGround();
    // Check to see if the first slice
    // drawn is whole in the X direction
    if (screenPosx % CELLSIZE)
    {
      // First and last slices are portions in the X axis.
      sliceNumber = SCREENSIZEX/CELLSIZE+1;
    }
    else
    {
      // All slices drawn are whole.
      sliceNumber = SCREENSIZEX/CELLSIZE;
    }
    firstSlice = screenPosx / CELLSIZE;
    drawCells( sliceNumber, firstSlice, INBACK);
    flipSurface();
}
```

Remember, the variable *screenPosx* is the index for your sliding window. Before you call drawCells(), you must determine what slice in the SLICE database corresponds to the current *screenPosx* value. The variable *firstSlice* is the slice index from where you'll start drawing the terrain tiles to the screen. From the *firstSlice* index you will draw a total of *sliceNumber* slices to the screen. The *sliceNumber* is the mapping of how large the DirectDraw surface is in SLICE units, which is SCREENSIZEX divided by CELLSIZE (640/32 = 20). The *sliceNumber* indicates that your sliding window is 20 slices wide. The *screenPosx* value, as you remember, is in screen coordinate units and is mapped to a SLICE index by dividing *screenPosx* by CELLSIZE. This is how you update your sliding window.

Moving Objects and Items

Good action and gameplay are important in designing an exciting game. I noticed another important design aspect in console-style games such as *Sonic the Hedgehog*, *Taz-Mania*, and *Captain America and the Avengers*, to name a few. They have a degree of curiosity. You, as the game player, are pulled into a world created by the game developer and there is that excitement of discovering what lies just ahead of you. In this section, I'll discuss how to add items and moving objects to your game that aren't a part of the virtual landscape. The simple techniques presented here will allow you to place mutant-seeking sentinel robots, bees that drop bombs, or lava falls in your own game design.

Enhancing the Scenery with Moving Clouds

In the CLOUDS example program, you take the TERRAIN example program and enhance it with moving clouds in the sky. You'll use some of the concepts and data structures used in the previous tile examples to add this simple effect. A moving cloud is a tile, but it's not fixed to a location on the terrain map.

Designing a Cloud CELL Database

You can use your CELL struct from your discussion on tiles to create a cloud tile database. This will contain information on how to retrieve the cloud cell from the offscreen DirectDraw surface, *lpDDSOne*, as shown in Figure 9-12. The code for the cloud cells database is shown in Listing 9-22.

LISTING 9-22 CELL Lookup Table for the cloud sprites

```
CELL    cloudCells[] = {
//X     Y       LEVEL       F   D       H           W
341,    482,    INFRONT,    0,  0,      63,         191,        // 1
341,    548,    INFRONT,    0,  0,      63,         191,        // 2
341,    614,    INFRONT,    0,  0,      63,         127,        // 3
};
```



Note that in reusing the CELL struct code you have changed the *beginLevel* and *endLevel* fields to represent the cloud tile height and width.

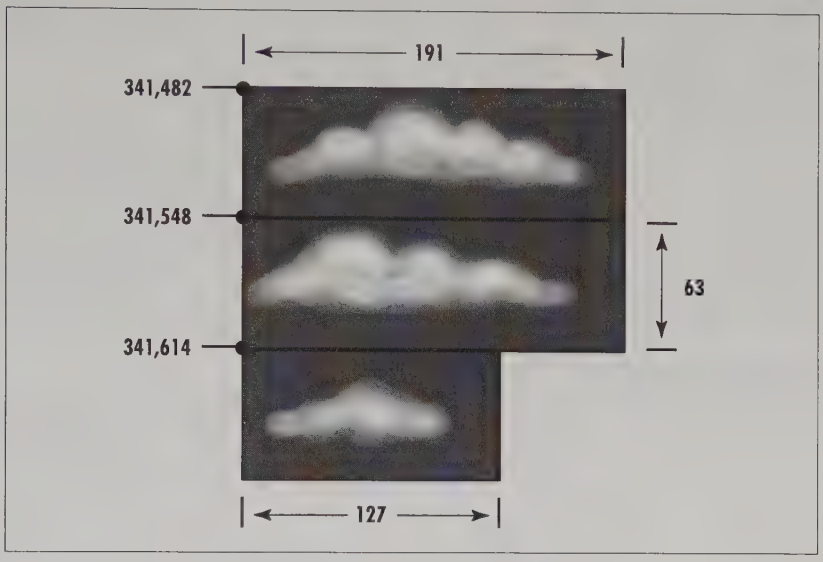


FIGURE 9-12
Using cloud cells to locate a cloud tile

With this code you have created a cloud cells database to locate the tiles on the offscreen DirectDraw surface. Now you need to provide the information as to where to draw the cloud tiles on the flipping DirectDraw surface, *lpDDSBack*. The CLOUDS example program uses the CLOUD structure, from the BUBONIC example program, to determine where the clouds are to be drawn on the DirectDraw surface. The CLOUD structure uses physical DirectDraw surface coordinates. The declaration of the CLOUD structure is shown in Listing 9-23.

LISTING 9-23 CLOUD structure used in *Bubonic the WartHog*

```
typedef struct {
  int     type;
  int     positionX;
  int     positionY;
  int     velocity;
}   CLOUD;
```

To move three clouds on the display, you will need to create an array of their *cloudCells* index, the initial *x* coordinates, the *y* coordinates, which will remain constant, and the cloud's horizontal velocities. Your simple array for the clouds is declared as shown in Listing 9-24.

LISTING 9-24 Clouds array with sprite location and cell velocity

```
#define NUMCLOUDS 3
CLOUD   cloudsINUMCLOUDSI = {
//type  positionX   positionY   Velocity
```

continued on next page

continued from previous page

```
0,      410,      30,      2,
1,      200,      70,      1,
2,      157,      45,      3,
};
```

That's it for your code declarations and array initializations. You now have clouds. What you need to do next is draw your clouds to the flipping DirectDraw surface, *lpDDSBack*. This will require modifying the function drawBackGround() from the TERRAIN example to add your ethereal effect.



Rendering the Clouds

The function drawBackGround(), presented here, is identical to the drawBackGround() used in the BUBONIC example code. When you draw the clouds you'll be updating the *x* coordinate (*clouds[i].positionX*) and holding the *y* coordinate (*clouds[i].positionY*) constant. The clouds move only in the horizontal direction from right to left on the display, on the flipping DirectDraw surface *lpDDSBack*. When they scroll off the left side of the display, at *clouds[i].positionX* <= 0, they wrap around to *clouds[i].positionX* = SCREENSIZEX -1, on the right side of the display. The code in drawBackGround() to handle these wraparounds is presented in Listing 9-25.

LISTING 9-25 Wrapping Clouds around to the beginning of the screen

```
rcRect.left = cloudCells[clouds[i].type].posx;
if( clouds[i].positionX < 0)
  rcRect.left -= clouds[i].positionX;
if (clouds[i].positionX + cloudCells[clouds[i].type].endLevel > 640)
  cloudOffset = SCREENSIZEX - clouds[i].positionX;
else if (clouds[i].positionX < 0)
  cloudOffset = cloudCells[clouds[i].type].endLevel + clouds[i].positionX;
else
  cloudOffset = cloudCells[clouds[i].type].endLevel;
```

The variable *cloudOffset* is used to smoothly wrap the cloud back over to the right side of the screen once it has completely scrolled off of the left. Remember, the *endLevel* value is the cloud sprite width and not a component of slope. With the proper cloud location calculated, you initialize the destination RECT, *rcRect*, with the proper values. The cloud is then drawn to the DirectDraw surface using the code shown in Listing 9-26.

LISTING 9-26 Drawing the clouds to the DirectDrawSurface

```
rcRect.right = rcRect.left + cloudOffset;
rcRect.top = cloudCells[clouds[i].type].posy;
rcRect.bottom = rcRect.top + cloudCells[clouds[i].type].beginLevel;
```

```
if (clouds[i].positionX < 0)
  cloudOffset = 0;
else
  cloudOffset = clouds[i].positionX;
drawOneToBack(cloudOffset, clouds[i].positionY,rcRect, FALSE);
clouds[i].positionX -= clouds[i].velocity;
if (clouds[i].positionX <= 0 - cloudCells[clouds[i].type].endLevel)
  clouds[i].positionX = SCREENSIZEX-1;
```

That's it for adding clouds. In a professional coding environment, it could probably take you a couple of hours to come up with the same thing. But it's worth the effort in the long run. The function used in CLOUDS and the BUBONIC examples are provided in Listing 9-27.

LISTING 9-27 The drawBackGround() function used in the CLOUDS example

```
void drawBackGround( void )
{
  RECT    rcRect;
  int     i;
  int     cloudOffset;

  // Restore background
  rcRect.left = 0;
  rcRect.top = TOPERASE;
  rcRect.right = SCREENSIZEX;
  rcRect.bottom = SCREENSIZEY;
  drawOneToBack(0, TOPERASE, rcRect, FALSE);

  if (!doClouds)
    return;

  i = 0;
  for ( i = 0; i < NUMCLOUDS; i++)
  {
    rcRect.left = cloudCells[clouds[i].type].posx;
    if( clouds[i].positionX < 0)
      rcRect.left -= clouds[i].positionX;
    if (clouds[i].positionX + cloudCells[clouds[i].type].endLevel
        > 640)
    {
          cloudOffset = SCREENSIZEX - clouds[i].positionX;
    }
    else if (clouds[i].positionX < 0)
    {
      cloudOffset = cloudCells[clouds[i].type].endLevel
                                        + clouds[i].positionX;
    }
    else
    {
```

continued on next page

continued from previous page

```
        cloudOffset = cloudCells[clouds[i].type].endLevel;
      }
      rcRect.right = rcRect.left + cloudOffset;
      rcRect.top = cloudCells[clouds[i].type].posy;
      rcRect.bottom = rcRect.top
                      + cloudCells[clouds[i].type].beginLevel;
      if (clouds[i].positionX < 0)
        cloudOffset = 0;
      else
        cloudOffset = clouds[i].positionX;
      drawOneToBack(cloudOffset, clouds[i].positionY,rcRect, FALSE);
      clouds[i].positionX -= clouds[i].velocity;

      if (clouds[i].positionX
          <= (0 - cloudCells[clouds[i].type].endLevel))
      {
        clouds[i].positionX = SCREENSIZEX-1;
      }
    }
  }
```

Be creative and draw whatever you want to for your background. The possibilities are endless. When I was playing around with this, I was also thinking how simple it would be to create a Windows 95 Direct X Game SDK version of *Frogger*. Remember that old game? The player maneuvers a frog across a river by jumping from log to log. It would be easy to make a log cells array and a simple frog. You could recycle most of the code from drawBackGround() to draw the logs in the river. Oh well, I guess that's a class assignment. The only way to learn is to go and play with this stuff.



Lists and Items

Another aspect of game design that makes a game interesting is having items the game character can pick up. These items can be bonus tokens for extra points: a shield, a weapon, health, or even super-fast sneakers to zip you to the end of the level. These components can be hidden to heighten the gameplay, or be easily accessible to the player. Hiding secret weapons and potions makes the gameplay more interesting. In this section, I'll discuss how to load items into a linked list and place items on the virtual terrain for the Bubonic character to pick up.



The Items List

The ITEMS example program and the BUBONIC demo game use the same linked list and code to generate the 3D rotating balls on the game screen. The

items are read in from a simple text file named ITEM.INF. The information in the file consists of three fields: the item type, the item's *x* coordinate, and the item's *y* coordinate. The example INF file has only three entries, but you could put 10, 20, 50, or 100 in if you desired. The format of the ITEM.INF file with the three entries looks like this:

```
0 200 200
0 300 300
0 900 250
```

Notice the *x* coordinate of the third item is 900. The DirectDraw surface, *lpDDSBack*, has a maximum width of 640 pixels. The item coordinates are using the same virtual terrain coordinates as the sliding window index, *screenPosx*, as shown in Figure 9-13.

Initializing the Items List

The idea behind the items list is that you can maintain one list per game level for the objects for Bubonic to pick up. The example uses a text file, ITEM.INF, but in a commercial game you would use a file database to retrieve the items and initialize the nth level of a game.

Some game design houses use C++ and OOP exclusively. Others use C++ and no OOP. In Chapter 8, you used an object-oriented linked list. In the ITEMS example code you'll use a standard linked list to process your items. If you're unfamiliar with list operations, go back and review Chapter 7. The

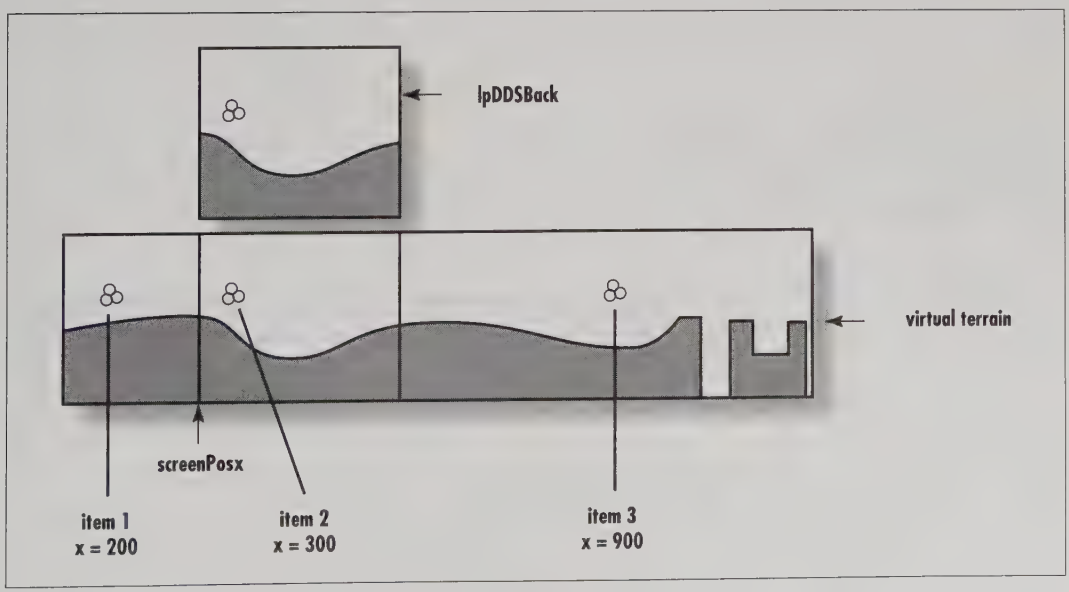


FIGURE 9-13 Virtual coordinates used by the items list

discussion in Chapter 7 focused on list objects but the concepts are identical. The standard list declaration used by ITEMS and BUBONIC is shown in Listing 9-28.

LISTING 9-28 The ITEM linked list declaration

```
typedef struct item_str {
    int         type;
    int         mapPosX;
    int         mapPosY;
    int         frameNum;
    DWORD       lastTickCount;
    int         speedX;
    int         speedY;
    item_str    *next_item;
}   ITEM;

ITEM            *item_head;
ITEM            *item_end;
int             numItems;
```

The items list is initialized by the function initItems(), which opens the ITEM.INF file, reads each tile entry, and places it onto the ITEM list. The code for initItems is shown in Listing 9-29.

LISTING 9-29 Initializing the ITEM list

```
BOOL initItems( void )
{
  FILE *stream;
  int  i;
  ITEM *item_ptr;

  // Open the terrain.inf file for reading.
  if(!(stream = fopen( "ITEM.INF", "r")))
  {
  return FALSE;
  }

  // Go through the file and initialize the item list.
  for (i = 0; i < NUMITEMS; i++)
  {
    if ((item_ptr = newItem()) == NULL)
      return FALSE;

    fscanf(stream,"%d",&item_ptr->type);
    fscanf(stream,"%d",&item_ptr->mapPosX);
    fscanf(stream,"%d\n",&item_ptr->mapPosY);
    item_ptr->frameNum = 0;
    item_ptr->speedX = 10;
    item_ptr->speedY = 0;
```

```
    item_ptr->lastTickCount = 0;

  }
  fclose(stream);
  return TRUE;
}
```

For this example, the number of items in the ITEM.INF file is defined in NUMITEMS. Using a loop, you read in all the item entries and insert them into the list. With each iteration of the loop, you allocate a place on the list. If the allocation is successful, function newItem() returns a pointer to the new list node. The code for function newItem() can be found in Listing 9-30.

LISTING 9-30 The newItem() function inserts items into the list function

```
ITEM  *newItem( void )
{
  ITEM    *temp_ptr;

  if (!(temp_ptr = (ITEM *)LocalAlloc( LPTR, sizeof(ITEM))))
    return NULL;
  if (item_head == NULL)
  {
    item_head = temp_ptr;
    item_end = temp_ptr;
  }
  else
  {
    item_end->next_item = temp_ptr;
    item_end = temp_ptr;
  }
  return temp_ptr;
}
```

The first field in the list node is the tile type. By now you know if you have a tile type field in your code; there is an associated CELL lookup table. Your *itemCells* table is shown in Listing 9-31.

LISTING 9-31 The ItemCells lookup table

```
CELL    itemCells[] = {
//X     Y       LEVEL       F   D       H           W
0,      719,    INFRONT,    40, 1,      16,         16,
};
```

In this table you have one entry for the items in the list. This means all the items are of the same type for this example. You can add your own items to this list and you may want to do so to experiment with how it works. In your example, the item is tile type 0. Its upper left corner is located in the offscreen DirectDraw surface, *lpDDSOne* at location (0,719). The blitting level is set to

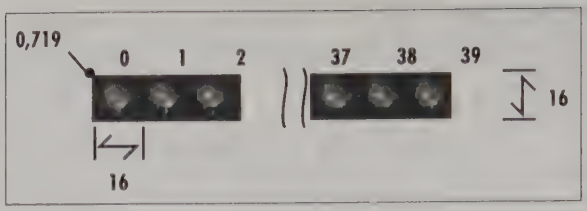


FIGURE 9-14
Items on the offscreen DirectDrawSurface

INFRONT, and there are 40 frames in this sequence of animation with a delay of 1 millisecond between frame changes. Each cell in the animation sequence is 16 x 16 pixels in size. For a graphical representation of the itemCells table refer to Figure 9-14.



Drawing the Items in the List

With the *itemCells* table and the linked list initialized, you are ready to draw your items to the display. When you draw the items, keep in mind that they are spread out across the virtual terrain. You identify which objects are to be drawn by using the sliding window's *screenPosx* value. If the *item_ptr->mapPosX* is within the range *screenPosx* and *screenPosx* plus 640 (the width of the flipping DirectDraw surface), the item(s) are drawn to the display. Another factor you must keep in mind when drawing the items is which frame is to be displayed. Remember your rotating molecule balls have 40 animation cells. Each frame has a duration of 1 millisecond before the frame index is incremented. The ITEMS example program and BUBONIC both use the function drawItems() for rendering the items on the list. The code for drawItems() is provided in Listing 9-32.

LISTING 9-32 The drawItems() function used to draw the list items

```
void drawItems( void )
{
  int     i;
  ITEM    *item_ptr;
  DWORD   thisTickCount;
  RECT    rcRect;
  int     xOffset, yOffset;

  thisTickCount = GetTickCount();
  item_ptr = item_head;

  // Cycle through the item list.
  for (i = 0; i < numItems; i++)
  {
    // Check to see if item is to be drawn at all.
    if (((UINT)item_ptr->mapPosX >= screenPosx)
```




```
      && ((UINT)item_ptr->mapPosX < screenPosx + 640))
    {
      // Item needs to be drawn.
      if((thisTickCount - item_ptr->lastTickCount)
              > (itemCells[item_ptr->type].delay))
      {
        item_ptr->lastTickCount = thisTickCount;
        item_ptr->frameNum++;
        item_ptr->frameNum %= itemCells[item_ptr->type].frames;
      }

      rcRect.left = itemCells[item_ptr->type].posx
      + (item_ptr->frameNum * itemCells[item_ptr->type].beginLevel);
      rcRect.right =
                rcRect.left + itemCells[item_ptr->type].beginLevel;
      rcRect.top = itemCells[item_ptr->type].posy;
      rcRect.bottom =
                rcRect.top + itemCells[item_ptr->type].beginLevel;
     xOffset = item_ptr->mapPosX - screenPosx;
     yOffset = item_ptr->mapPosY;
     drawOneToBack(xOffset, yOffset , rcRect, FALSE);
    }
    item_ptr = item_ptr->next_item;
  }
}
```

The function drawItems() cycles through the list using a loop to see if any items need to be drawn. If the *item_ptr->mapPosX* is within the bounds of *screenPosx* and *screenPosx* plus 640, the item is drawn. If the item cell needs to be drawn, check the *item_ptr->lastTickCount* field to see if one millisecond has passed since you last updated the frame index. Once the proper frame has been determined, you locate the desired cell of animation. The cell location offset is calculated as follows:

```
itemCells[i].posx + item_ptr->frameNum*itemCells[i].beginLevel;
```

Remember, you are recycling the *beginLevel* field. As shown here, it is the width of the item cell, 16 pixels, and has nothing to do with slope as does it with a terrain tile.



Item Collision Detection

The ITEMS example program demonstrated how to put items onto a list and draw the items to the display. In the *Bubonic the WartHog* demo program (BUBONIC), the warthog character runs around the terrain and picks up the rotating molecules for points. Like the items, the warthog uses virtual terrain coordinates to determine which items it may be picking up or colliding with. The function detectCollision(), in Listing 9-33, is used in *Bubonic the WartHog* for picking up items and incrementing the player's score.

LISTING 9-33 Collision detection routine used in *Bubonic the WartHog*

```
void detectCollision(void)
{
  ITEM    *temp_ptr;
  int     i;

  temp_ptr = item_head;

  for (i = 0; i < numItems; i++)
  {
    if (temp_ptr->mapPosX > player.mapPosX
       && temp_ptr->mapPosX < player.mapPosX + CHARSIZEX
       &&temp_ptr->mapPosY > player.mapPosY
       && temp_ptr->mapPosY < player.mapPosY + CHARSIZEY)
    {
       if (bSoundPlay)
         PanWAVInfo(lpwiGotItem);
       temp_ptr = destroyItem(temp_ptr);
       score += 100;
    }
    else
      temp_ptr = temp_ptr->next_item;
  }
}
```

You cycle through the list just as you did when you were drawing the items. You test the *item_ptr->mapPosX* (*temp_ptr* in Listing 9-33) and the *item_ptr->mapPosY* fields against the PLAYER struct player fields *mapPosX* and *mapPosY*. A bounding rectangle is constructed by the coordinates *player.mapPosX*, *player.mapPosY*, *player.mapPosX* + CHARSIZEX, and *player.mapPosY* + CHARSIZEY, where CHARSIZEX and CHARSIZEY are the height and width of Bubonic the WartHog, as is shown in Figure 9-15. If the item is within this

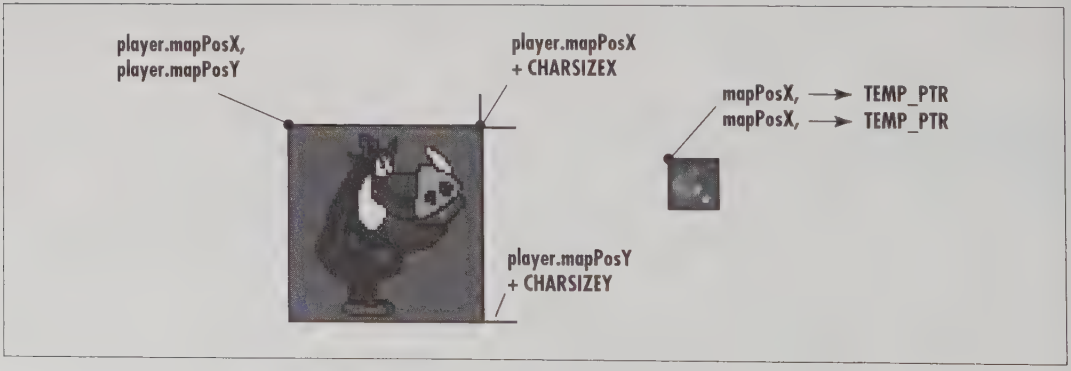


FIGURE 9-15 Collision detection in *Bubonic the WartHog*

bounding rectangle, a collision has occurred. The item referenced by *item_ptr* is then removed from the list, the score is incremented by 100, and DirectSound plays a WAV sound.

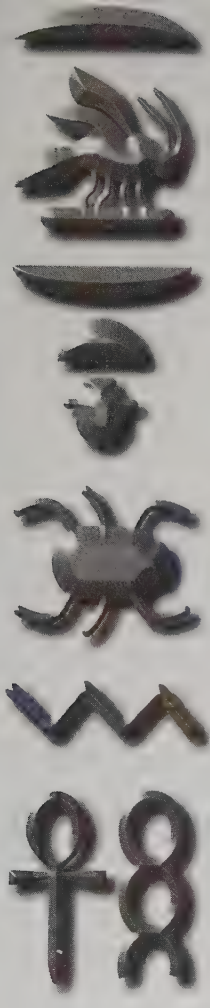


Conclusion

Bubonic the WartHog could easily be the subject of an entire book. Each aspect of a simple level has been broken down for you in the five example programs accompanying *Bubonic the WartHog*. The BUBONICB program was purposely designed to be simple. It has been left open for you to finish designing your own levels. There are some shareware games on the Internet, such as *JazzRabbit*. Download one of these games and analyze it. See if you can make similar critters and items in *Bubonic the WartHog*. Get some inspiration from other games and learn how to incorporate them in your own designs. Use your local pizza parlor for your own marketing surveys. Find out what coin-op game the kids like ("kids" is a relative, ageless term here), dump some quarters into it, and start reverse engineering.

10

DIRECTPLAY: NETWORKING IS THE KEY TO GLOBAL DOMINATION



DIRECTPLAY: NETWORKING IS THE KEY TO GLOBAL DOMINATION

For the longest time I measured my video gaming skills by reaching the highest level of a game and gaining satisfaction that I beat the computer. No matter what the computer used as its secret weapon, a purple monkey that throws barrels or menacing hulk robots out to destroy the last family on earth, the computer was able to provide an onslaught of attack waves, all of which I knew I could overcome by learning the tactics the enemies used at each level. When I went to sleep each night, I knew that rebel convoys made it to the moon of Endor, evil Nazi doctors had been blown to pieces, and Hades had failed in its attempt to take over my reality. All in all, the universe was a safer place because of my actions.

The day I set foot in a Virtual Reality World *MechWarrior* game pod, my method of measuring my gaming skills changed forever. At Virtual Reality World a group of players are turned loose on a networked 3D terrain. There is no computer opponent here; your opponents are in the pods next to yours. People, who five minutes ago were your friends, are now unloading auto-cannons and trying to blow apart your mech (a BattleTech Mechanized Armored Robot). The person you negotiated to be your attack partner before the game, has now also turned on you. The rules of the game have changed. The algorithm for this game is not predictable. You have no choice but to fire on anything that moves. You break free of this uncoerced attack and take cover behind a building. Now the hunt is on; your friends are out to get you, but not if you get them first.

Never before has your adrenaline pumped so hard in a video game. There is absolutely no way to predict the outcome of a human-controlled game. Computer opponents don't change sides on you or open fire on you from behind when you're being attacked by a missile-toting, demonically hideous monster. Developers can now provide this adrenaline rush to their market with the aid of the DirectPlay API.

EPISODE MEET THY MAKER



"13, are you copying any of this?" I asked my young friend.

"Roger that, Bary. It looks like you hit the jackpot."

"Yeah. Has Kala come around yet? I sure could use her help on this one. This looks like Agamemnon's operations center. Or virtual operations center, I should say. Take a look at the control console in the center of the room."

"I'm on it, Bary. It's definitely a command console. But I can't make out any of the inscriptions on it."

"Do you think it's alien, 13?"

"Negative. This is definitely ancient Egyptian but I'm not versed in the language or culture. I wish Kala were conscious at this point. She's our expert."

"I'm going to pan the room so you can get a look at this place. There are five doors leaving this chamber. Each is located at the apex of a polygon inscribed on the floor. It's a pentagram with the command console located at the center."

"Be careful in there; I think we located the heart of the artifact. We encountered insurmountable odds to get this far. It's a safe bet this place is booby trapped," 13 added.

"13, I'm going to pan back to the doors. I don't know if you saw that image flash. I just thought I saw a Russian Soyuz rocket take off like I was watching a video transmission. It flashed and in the door next to it I saw what looked like the Japanese stock market."

"This doesn't look good, Bary."

"Tell me about it. I think our friend, the Grinch, provided Agamemnon with a network interface. We've got to destroy this command center. Agamemnon can literally run the world from here. I'm going to pan over to a door on the right. There's a display of the entire northwestern telecommunications grid. He's breaking the security codes of phone company circuit offices and taking control without anybody noticing."

"Bary, I'm reading a power surge. Agamemnon's chamber of souls is adjacent to your present location. He's summoning his army. The hallways are filling with sprite blips. You're about to have company. Seal off that room any way you can!"

"Please tell me Kala is slightly awake, 13."

"Sorry Bary; she's sleeping like a baby."

"Okay 13. We've got nothing else to lose; let's start pressing buttons."

"I don't recommend that choice of action, Bary."

"What should I do? Wait here till you come up with an idea?"

"You're correct, Bary. An act of desperation is the only suitable action we can take at this moment."

"I am going to press the big red flashing button over here on the panel to my left. In the science fiction movies, this is always the self-destruct button."

I press the red button and wait a few seconds.

"13, do you see any change, like symbols counting from ten backwards?"

"Negative. There is another power surge. Energy is being directed to your location on the console."

I look around the room and see no change in any of the virtual portal doors or console displays. All is quiet. After a few seconds, a cylindrical burst of light shoots up from the center of the console. Thousands of polygon lines appear. As soon as the polygons are organized, a texture is mapped over the polygons to create a face. I recognize the face. It is the same face that was used to create the golden death mask on the artifact. And now I'm face to face with it.

"Bary Maelstrom, you have defiled my sacred chamber. For that you will serve me forever!"

"Gulp!"



Networks and Internetworking

In the classic sense, a network is two or more computers *physically* connected to a single piece of copper wire. Computers communicating with one another along this piece of wire are part of the network. A popular network connection method, developed at Xerox Parc in the early 1970s, is called an Ethernet network. Data on an Ethernet network is divided into pieces called packets and transmitted through the network and is called packet switching. Computers on a packet-switched Ethernet identify each other with 48-bit (6-byte) addresses. Multiple Ethernet networks are joined by network gateways called routers. When two or more networks are joined by a router, an internetwork takes shape; see Figure 10-1.

Packet-switched Ethernet networks are called local area networks (LANs). LANs may be joined to remote networks across town, in another state, or around the world through circuit-switched networks, or what we refer to as the phone company. The connection of two LANs through phone lines, a switching office, or trunk lines such as T1s, T3s, or ISDN, is called a wide area network (WAN). The term *network* usually means a packet-switched network such as a LAN. Figure 10-2 shows a simple WAN configuration connecting a game developer in Los Angeles with a game tester in Dallas.

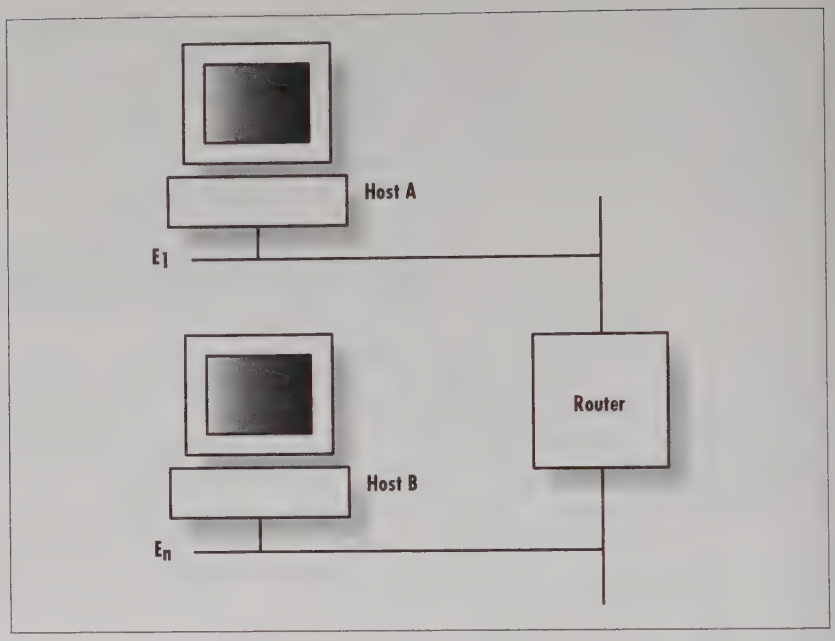


FIGURE 10-1
Two Ethernet segments form an internetwork

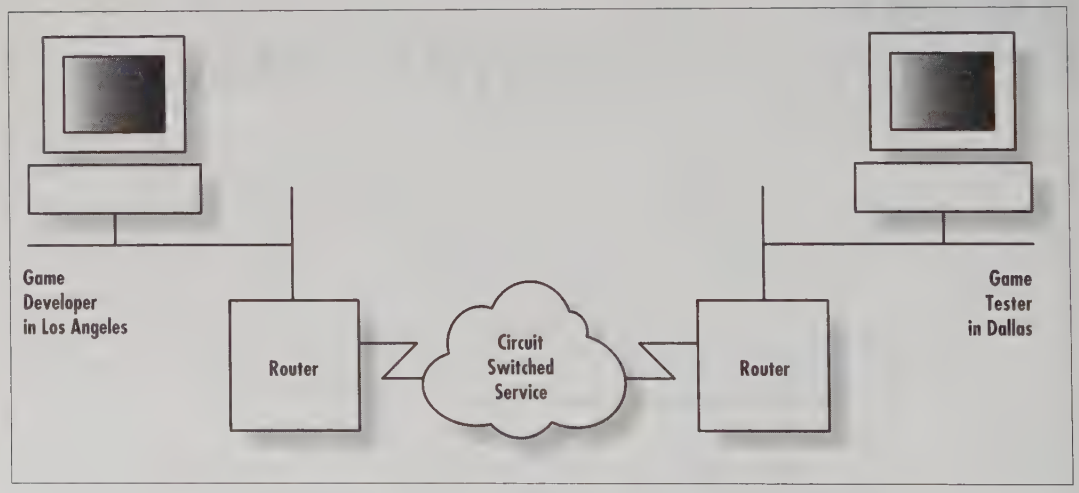


FIGURE 10-2 A simple WAN

The figure shows that routers at each site are connected by serial links into a circuit switched service. Network professionals call this connection a *cloud* because data is handed off to the phone company and it mysteriously reappears

on the router at the other side. The game developer in Los Angeles and the software tester in Dallas are internetworked. These two computers can communicate with each other because the computers follow network messaging standards called protocols.

Network Layers and Protocols

Data communication with network protocols is vendor-independent. This means a Macintosh computer with a Farallon network card can communicate with an IBM PC using a 3Com card. The two computers don't care about each other's hardware or network cards. They have agreed to send their data in a specific byte order and obey communication protocols for sending and receiving their information.

Networking protocols are organized in layers, shown in Figure 10-3, called the Four Layer Network Model. The Seven Layer OSI Model exists but pretty much has been abandoned by the industry. Transport Control Protocol/Internet Protocol (TCP/IP) suite is a group of protocols that use this model. Each network layer provides a service to the layer above it. TCP/IP is a combination of protocols at different layers of the network model.

The Application Layer

The very top layer is the application layer. Most people with e-mail use this layer on a daily basis. The application layer is host to application-specific protocols such as Simple Mail Transfer Protocol (SMTP), telnet, File Transfer Protocol (FTP), and Simple Network Management Protocol (SNMP). This is the application interface layer. For those of you with Web browsers, your browser uses the application layer with a telnet port 80 connection.

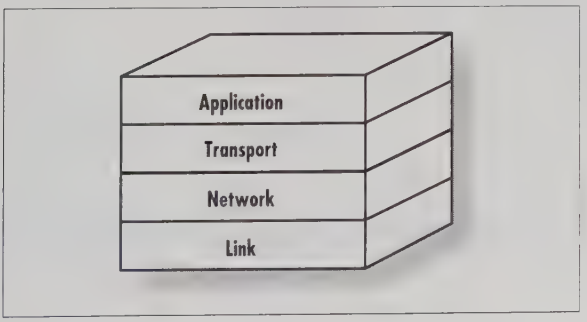


FIGURE 10-3
The four layers of the TCP/IP protocol suite

The Transport Layer

The layer beneath the application layer is the transport layer. The role of the transport layer is to provide communication from one application program to another. In using the TCP/IP suite there are two transport protocols: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). These two protocols are responsible for the flow of data between two computers. Both protocols divide the data to be transmitted into chunks of data called packets. TCP checks to make sure that the packet was received properly by the host at the other end. Because of this, TCP is called a reliable transport. UDP, on the other hand, sends its packet, called a *datagram*, out to the host without ensuring its delivery. TCP is commonly used in a client-server environment where a user is making queries on a large database. Both TCP and UDP are supported by Microsoft's WinSock interface. The DirectPlay API currently only supports UDP transport over WinSock using IP.

The Network Layer

The network layer handles the communication between networked computers. A request is made at the transport layer to deliver a datagram to a specific machine with an Internet Protocol (IP) address. Now that most people are Internet savvy with World Wide Web browsers, they are familiar with the IP octet addressing convention. It consists of four bytes (32 bits) with each byte separated by a decimal point, such as 128.16.33.107. This is called an IP address and is used by the network layer to transmit a datagram. Using this IP address, the TCP or UDP packet is encapsulated by an IP datagram and sent out into the network (see Figure 10-4). Encapsulation is the mechanism by which one layer hands a datagram to another. When the IP datagram reaches its destination machine, the network layer strips off the IP datagram and hands the TCP or UDP packet up to the transport layer.

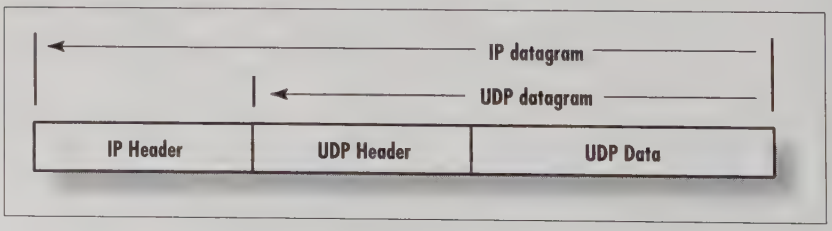


FIGURE 10-4
Encapsulation of a UDP datagram

The Link Layer

The link layer is the lowest level of the model. This is the network interface card (NIC) and its device drivers. This is the layer responsible for transmitting and accepting IP datagrams. This is where your computer is physically connected to the network, whether it's through an RJ-45 connector, AUI adapter, or coaxial cable.

At the link layer are several different kinds of transmission mediums: Ethernet, token ring, Fiber Distributed Data Interface (FDDI), ISDN, and RS-232, to name a few. The most familiar of these terms is Ethernet. Professional developers call this the IEEE 802.3 standard for Ethernet, operating at 10Mps with 48-bit hardware addresses. Ethernet encapsulates IP datagrams in packets called frames. Figure 10-5 shows a UDP datagram encapsulated all the way down the four layers for transmission on the Ethernet.

ARP...Excuse Me

The network layer uses 32-bit IP addresses and the hardware layer has 48-bit addresses. The 32-bit IP address is an abstraction that the network and transport layers understand. The physical cable can have more protocols than just IP. On a typical corporate net you'll find AppleTalk, DECNet, Novell, and possibly Vines to name a few. In the cable, hardware destinations are determined by 48-bit hardware addresses. Once the frames reach the right host, it is then brought inside and sent up to the network layer for processing.

What can you do if you want to map a 32-bit IP address to a 48-bit hardware address? The TCP/IP protocol suite allows you to do this with Address Resolution Protocol (ARP), a service provided between the network

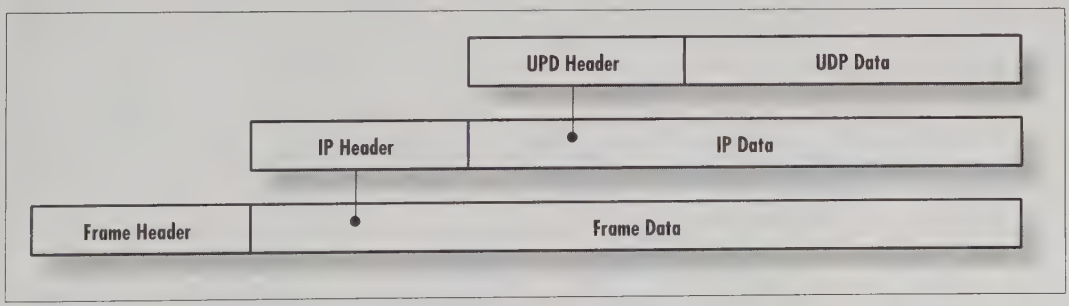


FIGURE 10-5 Encapsulation of a UDP datagram for Ethernet transmission

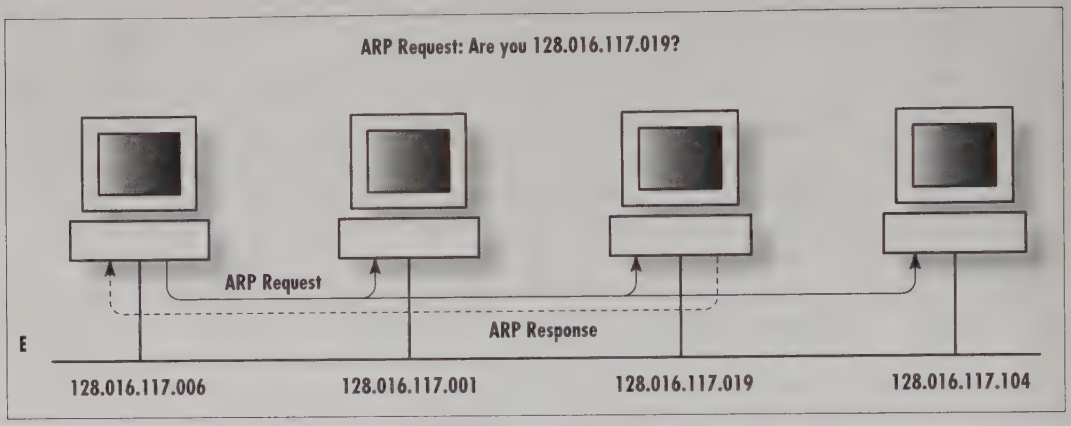


FIGURE 10-6 ARP requests and responses

and hardware layers. ARP works by sending a message to every host on the network, called a broadcast, asking them if they own this IP address; see Figure 10-6. If a device responds to the ARP, the IP address as well as the hardware address are then maintained in ARP cache. The mapping in the cache is finite and expires in less than half an hour. ARP provides dynamic mapping between IP and hardware addresses. ARP cache entries allow you to send a frame directly to a hardware address that corresponds to the IP address. From there the frame can be stripped and the IP packet can be delivered up the layers to its destination. For those of you who are insomniacs, the Request For Comment (RFC) 826 contains all the specifications on ARP.



User Datagram Protocol (UDP)

Two distinct methods of connection exist in a client-server environment: connection-oriented and connectionless server management. In a connection-oriented approach, the client requests a connection to a server and the connection is opened between the client and the server. Data is sent across the connection and its packet size and content are validated. If a packet fails to reach its destination, it is re-sent. This is known as a reliable transmission. When the transactions are through, the connection is closed. TCP is a connection-oriented protocol.

UDP is a *connectionless* approach to client-server architecture. There is no checking to validate reliable transmission. Typically, the client side is responsible for data delivery. Extra messaging must be included in connectionless environments to determine if a retransmission is necessary. This is a subtle hint to software development managers that true Internet messaging is complex and shouldn't be left in the hands of college interns. The simplicity of the DirectPlay

architecture makes the messaging look easy, but Internet messaging requires a professional Internet savvy programmer. Furthermore, commercial developers should always test their code with network traffic generators on secure nets. Create as much of a protocol soup on your lab net as possible, then throw in the AppleTalk, IPX, and DECNet protocols. If at all possible, get a serial data channel traffic generator to simulate frame relay and HDLC traffic. Programmers who have worked with time-to-live values, and timeouts on the Internet over circuit switches, can tell you that the Internet is a cruel mistress!

UDP datagrams are simple but provide no reliability. The application layer requests to send a message to the network. UDP encapsulates the message and sends it to the network layer with no guarantees it will make the destination address. Another concern is IP fragmentation over the net. In our ideal world of lab networks the DirectPlay messages are limited to 256 bytes in size. Fragmentation in the lab isn't really an issue, especially with messages (datagrams) this small. However, on the Internet we have no idea how a datagram is going to arrive to its destination. One datagram can leave our computer and experience seven router network hops to get to its destination through a cloud. A second packet could leave our system immediately after and experience only four router network hops (for some strange reason) and beat the previous packet to its destination. Such a situation could arise over leased lines.



Internet Packet eXchange (IPX)

Another connectionless transport protocol available to DirectPlay through the WinSock interface is IPX. This protocol belongs to the Novell NetWare IPX/SPX protocol suite similar to TCP/IP. In most corporate PC configurations, both TCP/IP and IPX are available on the system concurrently running multiple protocol stacks. IPX evolved from the Xerox Network Standard (XNS), which is the Xerox networking protocol. A Windows 95 user with a network card can activate IPX as a transport protocol by enabling NetBIOS in their network settings on their control panel. This won't allow them to log on to a Novell server, but it will activate the necessary drivers to use WinSock over IPX for DirectPlay usage.

There are some subtle differences in how UDP/IP and IPX behave. In the TCP/IP protocol suite, UDP is the transport layer and it provides a 32-bit IP address to the network layer for IP to deliver the datagram. If the IP address mapping does not exist in the ARP cache, the IP sends out an ARP request to determine the destination IP's hardware address. In the NetWare solution, the transport (UDP) and network (IP) layers have been fused into one functionality. IPX uses 10-byte long network addresses which consist of a 4-byte (32 bits) network number and a 6-byte (48 bits) node address called a Media Access Control (MAC) address. The 48-bit MAC address is burned into the ROM on your network interface card. The 32-bit network number has no correlation to

the 32-bit IP address. This 32-bit value is assigned to the client by the Novell server. This address is a hexadecimal address and does not use the IP dotted octet address format.

Interoperabitlity between IPX and TCP/IP is provided with the Microsoft WinSock interface. This allows both TCP/IP protocol stacks and NetBios IPX stacks to exist simultaneously in your system. This doesn't mean one game player can select IP and the other player select IPX and expect to see a game session available. Both players must select the same transport medium, either IPX or IP. Although this scenario has sparked my interest in designing a master game server where the clients can come in on any medium they want, it would allow for a pretty cool role playing game engine where IPX and TCP/IP players could attach.



DirectPlay and Interprocess Communication

After seeing how protocols behave in the network four-layer model, you're ready to focus on how DirectPlay programs communicate using these protocols. Much of the attention here will focus on DirectPlay sessions over IP and how the games are handled in a client-server environment.



Ports

More than one process, at any one time, can be using the transport layer with TCP or UDP. The Windows 95 operating system could be blasting mutants in a DirectPlay game that runs over IP, playing against a coworker in the next cube. While the game is going on, e-mail could be coming in from the Internet. The transport layer identifies processes in the application layer by port numbers. These are 16-bit integers that tell the TCP and UDP datagrams where data is to be sent. E-mail (SMTP) uses TCP port 25, telnet is port 23, the Web port is 80, and TFTP is on UDP port 69. If you wanted, you could send your coworker an e-mail message from God by telnet-ing into your mail server using telnet [destination ip address] 25.

Ports 1 through 1023 are reserved on all TCP/IP accessible computers. These are called *well-known* port numbers. As demonstrated with the e-mail from God example, accessing ports requires a 32-bit IP address of the host system the desired service is on and a 16-bit (positive) port ID of the service being requested.

During a DirectPlay session, the client game needs to communicate through a port. The game is at the application layer sending messages down to the transport layer, which is encapsulated again and sent down to the network layer to be delivered to the game server. The client port number is primarily for providing a data transaction point for the client game. It is merely a unique

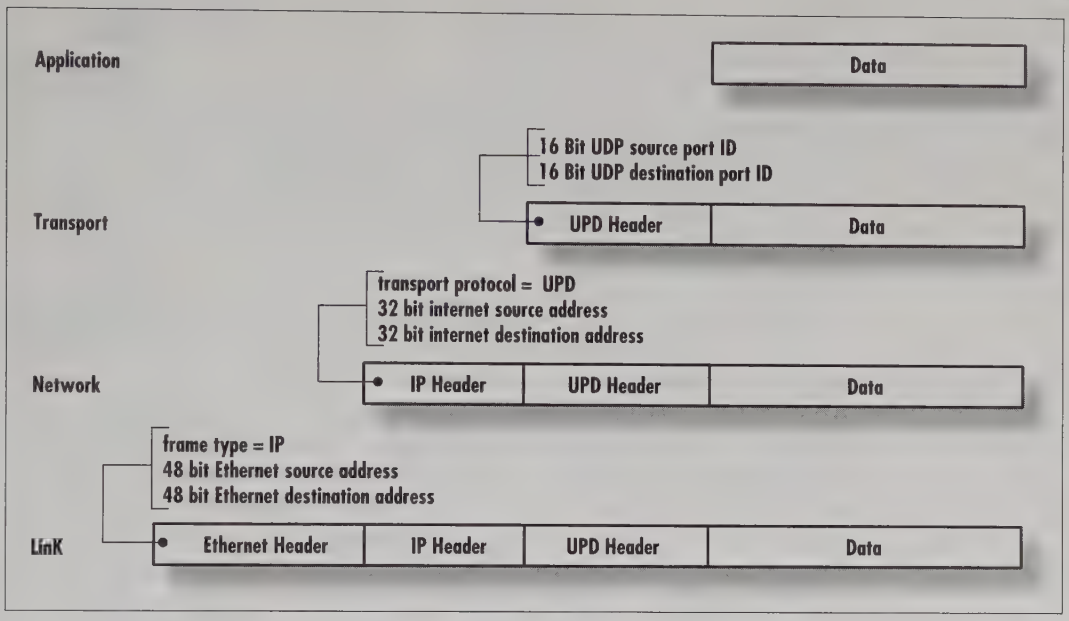


FIGURE 10-7 Encapsulation of a UDP datagram and its port IDs

16-bit integer whose value is greater than 1023. This port ID is called ephemeral, or short-lived, because the client port ID only exists as long as the application is executing. Figure 10-7 shows a UDP datagram with its source and destination port assignments.

By using a network sniffer, you can observe the behavior of an IP DirectPlay session. It appears that the client consistently sends out a UDP datagram broadcast advertising port 8787. The server listens for this broadcast and returns a port value back to the client. The port ID returned by the server is the port ID that the client and the server use for the duration of the DirectPlay session.



Sockets

In the early 1980s, the Advanced Research Projects Agency (ARPA) funded a project for the University of California at Berkeley to port TCP/IP over to UNIX. The group was asked, as part of the port, to add system calls to support TCP/IP functionality. Early TCP/IP ports to UNIX used I/O style system calls, such as open, read, write, and close. The designers decided that in order to encompass multiprotocol support, such as XNS, which was popular at that time, they would have to abandon this simple solution. What the group at Berkeley came up with was an interface that used UNIX-like system calls wherever possible and added new calls to support protocols such as TCP/IP and

XNS. The calls that support the TCP/IP interface are known as a *socket interface*. It was first released in BSD UNIX 4.1.

In BSD UNIX, each socket created is assigned an integer value called a socket descriptor. Socket descriptors reside in the same descriptor table as file descriptors. The system function socket() creates a socket as opposed to the UNIX I/O call open() which creates a file descriptor. Therefore, one system call creates a socket and the application uses other system calls to specify the socket interface usage.

When an application requests a socket, the UNIX operating system allocates memory for the socket data structure and assigns a descriptor table entry to the data structure. Figure 10-8 shows a descriptor table with a socket descriptor table entry.

The call to socket() requires the specification of a protocol family. Most implementations of socket() only support the PF_INET. WinSock uses AF_INET as its protocol family specifier. From Figure 10-8, the service SOCK_DGRAM was selected. This selects UDP datagrams as would be used in DirectPlay. If the macro SOCK_STREAM is used, TCP would have been selected. As you can see, the call to socket() leaves most of the other fields uninitialiazed. These fields are filled in later when specifying which host and port to send a datagram to.

Once a socket is created, it has two functions. It can either initiate a connection, or wait for a connection. A server typically waits for a connection from a client. A socket used by a server that waits for a connection is called a *passive* socket. Clients, on the other hand, establish connections to servers. A socket used by a client initiating a connection is called an *active* socket.

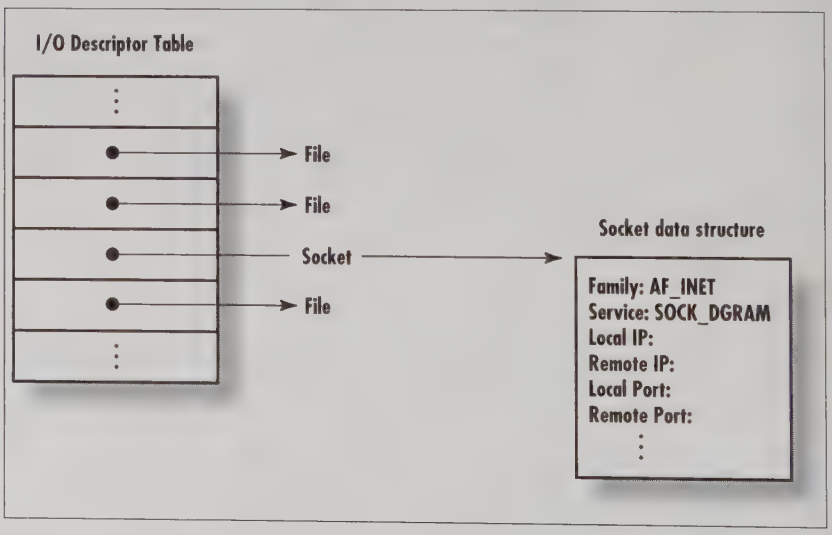


FIGURE 10-8
Descriptor entry for a socket data structure



WinSock

The Windows Socket Application Programming Interface (WinSock API) is the library for Microsoft Windows to provide compatibility with the BSD UNIX socket implementation. WinSock supports the TCP/IP protocol suite as well as IPX/SPX and is expected to support AppleTalk.



Blocking

BSD UNIX socket functions do not return to the calling program until the operation is completed. This mode of software operation is called blocking. An example of a socket function that will block is the recv() call. This function will block until it receives a response from the host. The notion of blocking is that the function does not return until it has completed its task. BSD UNIX is a multitasking operating system and this doesn't usually present a problem. In Windows 3.1, however, it's not a good programming practice to call a blocking operation in a message-based operating system. The WinSock library was designed with a set of blocking and non-blocking socket calls. The term *asynchronous* is used by Microsoft to describe non-blocking functions. Functions like WSAAsyncGetHostByName() are the asynchronous function for the BSD UNIX routine gethostbyname().

In Windows, WinSock calls are handled by winsock.dll. The DLL will initiate the function when it is called and execute a message loop for dispatching and receiving. This allows other threads to execute if the WinSock operation is idle. However, WinSock doesn't currently have provisions to execute a second WinSock call if it is currently in a blocking mode. The WinSock specification states that the programmer should use the WinSock routine WSAIsBlocking() to check if a WinSock call is in progress.

Why do you care about the functionality of WinSock if DirectPlay does this all for you? When debugging the DirectPlay code there are two places to look for answers: one is the network sniffer and the other is the workbench debugger. Traces will reveal nothing from DirectPlay. In the network model, DirectPlay is actually an application-level library. You'll be looking for UDP and NetWare IPX packets on the sniffer. The debugger screen will tell you nothing more than if you made the call correctly. A clear understanding of TCP/IP, networks, socket, and WinSock library routines is imperative to the commercial developer for creating a successful DirectPlay application.

The DirectPlay API

The DirectPlay API, simply put, is the Microsoft Windows 95 game connectivity API. The DirectPlay interface consists of a service-providing medium and a DirectPlay object. The service-providing medium is the method of connectivity between the remote game systems. Service providers can be online services or networked services. DirectPlay API comes with service providers for dial-up and also IP and IPX over the WinSock API.

The DirectPlay object runs on top of the service provider, the communication link. Once the DirectPlay object has been instantiated, games can communicate using Send() and Receive() messaging functions. The DirectPlay interfaces use the Component Object Model discussed in Chapter 3.

Initializing a DirectPlay Session

In previous DirectX objects, such as DirectDraw and DirectSound, the objects were instantiated through the DirectDrawCreate() or DirectSoundCreate() interfaces. Instantiating a DirectPlay object is not as simple. Before a DirectPlay object can be created, the user must first choose a medium for the session. The medium is the connectivity layer between the client and the game server. There are two distinct abstractions in the DirectPlay session, as Figure 10-9 shows. There is the DirectPlay instance and there is the connectivity medium, called the Service Provider.

The session's service provider can be a modem-to-modem connection, a dial-up service such as CompuServe, AOL, the Microsoft Network, or a network connection running over IP or IPX. The examples in this book will use the service providers for network services such as IP (WinSock) and IPX (WinSock).

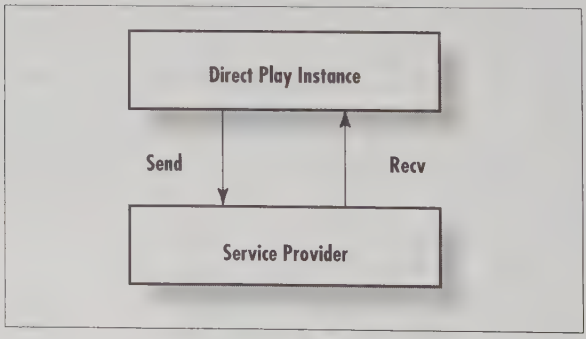


FIGURE 10-9
A DirectPlay instance and a service provider

Initializing the Session Provider

To instantiate a DirectPlay object, you must know the DirectPlay service medium. Currently, DirectPlay has device drivers for modem play and network play using IP and IPX transport protocol. Programmers have no idea what the players are going to select for their session provider. Programmers can provide a list of services available for a DirectPlay session and allow the user to select from this list. DirectPlay provides a function for the programmer to ask the Windows 95 operating system what DirectPlay drivers it has installed. An enumerated list of session providers can be obtained by calling the DirectPlay API routine DirectPlayEnumerate(). This routine requires a CALLBACK function that is used by the Windows 95 operating system to provide an enumerated list of DirectPlay device drivers available. The prototype for both the DirectPlayEnumerate() and the CALLBACK function are shown in Figure 10-10. An explanation of CALLBACK functions can be found in Chapter 1.

The enumerated list returned by the CALLBACK function can be displayed in a Windows 95 dialog list box. From the list box, users can select which provider they wish to use for their DirectPlay game session. The resource file DPTOOLS.RC has a simple barebones dialog list box, resource ID IDD_CHOOSEPROVIDER, that can be used for this purpose. The CALLBACK function, DlgProvider() in source file DPTOOLS.CPP, demonstrates how to generate a service provider list with DirectPlayEnumerate(). The function DlgProvider() does two things. First, it generates an enumerated list box of session providers available. Second, it instantiates a DirectPlay object based on the session provider the user selected. The dialog list box used by DlgProvider() is shown in Figure 10-11.

The first action to accomplish when the dialog box is displayed is to have the list box filled in with the enumerated listing of session providers available. To generate the list, handle the Windows message WM_INITDIALOG, which

```
HRESULT   DirectPlay Enumerate(
          LPDPENUMCALLBACK   lpCallback,
          LPVOID             lpContext)

BOOL      Callback (
          LPGUID    lpGuid,
          LPSTR     lp Driver Description,
          DWORD     dwMajorVersion,
          SWORD     dwMinorVersion,
          LPVOID    lpContext)
```

FIGURE 10-10
The DirectPlay API routine to list service providers

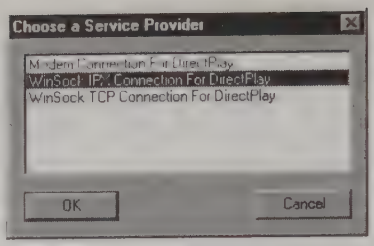


FIGURE 10-11
Dialog box used by the CALLBACK function DlgProvider()

is sent just before displaying the dialog box. Listing 10-1 shows the code to handle the Windows message WM_INITDIALOG.

LISTING 10-1 Initializing the session provider dialog

```
case WM_INITDIALOG:
  hWndCtl = GetDlgItem( hDlg, IDC_LIST1);
  if (hWndCtl == NULL)
  {
    EndDialog( hDlg, TRUE );
    return TRUE;
  }

  // add service providers to list box
  DirectPlayEnumerate(GetDPServices, (LPVOID) hWndCtl);
  SetFocus(hWndCtl);
  SendMessage(hWndCtl, LB_SETCURSEL, 0,0);
  return FALSE;
```

Use the Windows API call GetDlgItem() to display the resource list box IDC_LIST1 from DPTOOLS.RC. If the call is successful, call the DirectPlay API routine DirectPlayEnumerate() and pass a pointer to the CALLBACK function GetDPServices(). The function name GetDPServices() is arbitrary. This is what I used in source file DPTOOLS.CPP. You can call yours, Fred(), Wilma(), GetDaStuff(), or whatever. What you have to pay careful attention to is the function prototype and how the CALLBACK function behaves. The CALLBACK function prototype is given in Figure 10-10. This is the function used by Windows 95 to display the session providers in the dialog list box when it queries DirectPlay about what services are available. The CALLBACK function basically tells Windows 95, based on the list it found, to add the description field strings using message LB_ADDSTRING, to the list box. The message LB_SETCURSEL tells Windows to scroll the listings if they exceed what can be displayed. The CALLBACK function passed to DirectPlayEnumerate() is shown in Listing 10-2.

LISTING 10-2 Selecting a session provider

```
BOOL FAR PASCAL GetDPServices( LPGUID lpGuid, LPSTR lpDesc,
                               DWORD dwMajorVersion,
```

```
                                    DWORD dwMinorVersion, LPVOID lpv )
{
  LONG iIndex;
  HWND hWnd = (HWND) lpv;

  iIndex = SendMessage(hWnd, LB_ADDSTRING, 0, (LPARAM) lpDesc);
  if (iIndex != LB_ERR)
    SendMessage(hWnd, LB_SETITEMDATA, iIndex, (LPARAM)lpGuid);

  SetFocus(hWnd);
  SendMessage(hWnd, LB_SETCURSEL,0,0);
    return TRUE;
}
```

The parameters being passed to the function contain descriptive information about each device driver being enumerated in the dialog box. For instance, the parameter *lpGuid* is the pointer to the GUID of the device driver. The *dwMajorVersion* and *dwMinorVersion* fields contain information about the revisions of the specific driver. The field *lpDesc* is the description string that is to be displayed in the dialog's list box.

At this point, there is a mountain of dialog code that does nothing more than generate a dialog box (resource ID IDD_CHOOSEPROVIDER) and fills in its list box, (resource ID IDC_LIST1) with an enumerated list of DirectPlay session providers. You have triggered off of the Windows 95 message, WM_INITDIALOG. All of the efforts to this point have been for dialog box initialization.

When the dialog list box is displayed, wait for a WM_COMMAND message to notify you of the user's selection. The user interface provides a mechanism for the user to make a selection. They may use the keyboard to scroll up and down the list, using the arrow keys. The user may use the mouse and double-click on a selection. Or they may use the OK and Cancel control buttons. For this example, let the user select the desired medium by double-clicking on the list box. The code for this is shown in Listing 10-3.

LISTING 10-3 Dialog controls for selecting the session provider

```
case WM_COMMAND:
  switch(HIWORD(wParam))
  {
    case LBN_SELCHANGE:
      iIndex = SendMessage((HWND) lParam, LB_GETCURSEL,0,0);
      hWndCtl = (HWND)lParam;
      return FALSE;

    case LBN_DBLCLK:
      iIndex = SendMessage((HWND) lParam, LB_GETCURSEL,0,0);
      if (iIndex != LB_ERR)
      {
        lpGuid = (LPGUID) SendMessage((HWND) lParam,
```

continued on next page

continued from previous page

```
        LB_GETITEMDATA, iIndex, 0);
        DirectPlayCreate(lpGuid, &lpDP, NULL);
        EndDialog(hDlg, TRUE);
        return TRUE;
      }
      break;
  } // switch hiword
```

The notification message LBN_DBLCLK tells the DirectPlay session launcher that the user has selected a session provider. The pointer to the GUID of the session provider's device driver is then returned. This pointer can now instantiate the DirectPlay object with the DirectPlay API call DirectPlayCreate(). Its prototype is shown in Figure 10-12. Following the instantiation of the DirectPlay object, the dialog terminates and returns to the parent application.

The complete source for the service provider dialog is presented in Listing 10-4.

LISTING 10-4 Functions used for selecting the session provider

```
int GetProvider(HWND hwnd)
{
  return (DialogBox(NULL,(LPCSTR)IDD_CHOOSEPROVIDER, hwnd,
            (DLGPROC) DlgProvider));
}

BOOL CALLBACK DlgProvider( HWND hDlg, UINT msg,
                           WPARAM, LPARAM )
{
  LPGUID lpGuid;
  static long iIndex;
  static HWND hWndCtl;

  switch (msg)
  {
    case WM_INITDIALOG:
      hWndCtl = GetDlgItem( hDlg, IDC_LIST1);
      if (hWndCtl == NULL)
      {
        EndDialog( hDlg, TRUE );
        MessageBox( GetActiveWindow(),"Provider
                        Failed","ERROR", MB_OK);
        return TRUE;
      }
      // add service providers to list box
      DirectPlayEnumerate(GetDPServices, (LPVOID) hWndCtl);
      SetFocus(hWndCtl);
      SendMessage(hWndCtl, LB_SETCURSEL, 0,0);
      return FALSE;

    case WM_COMMAND:
      switch(HIWORD(wParam))
      {
```

```
        case LBN_SELCHANGE:
          iIndex = SendMessage((HWND) lParam, LB_GETCURSEL,0,0);
          hWndCtl = (HWND)lParam;
          return FALSE;

        case LBN_DBLCLK:
          iIndex = SendMessage((HWND) lParam, LB_GETCURSEL,0,0);
          if (iIndex != LB_ERR)
          {
            lpGuid = (LPGUID) SendMessage((HWND) lParam,
                              LB_GETITEMDATA, iIndex, 0);
            DirectPlayCreate(lpGuid, &lpDP, NULL);
            EndDialog(hDlg, TRUE);
            return TRUE;
          }
          break;
      } // switch loword

      switch(LOWORD(wParam))
      {
        case IDCANCEL:
          EndDialog( hDlg, FALSE);
          return TRUE;
        case IDOK:
          EndDialog( hDlg, FALSE);
          return TRUE;
      } // switch loword
      break;
  } // switch msg
  return FALSE;
}
```

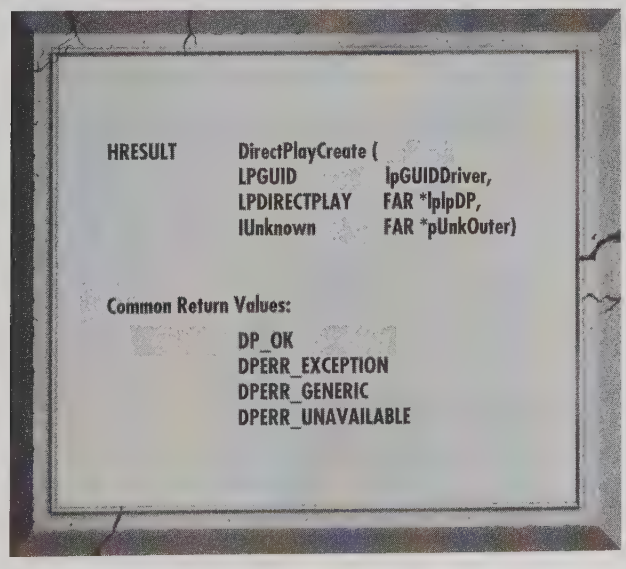


FIGURE 10-12

The DirectPlay API call DirectPlayCreate()

Getting the Service Provider in Review

Now there is a DirectPlay object. The DirectPlay object was created by retrieving a list of DirectPlay service providers from the Windows 95 operating system using a CALLBACK function. An enumerated list of device drivers supported by DirectPlay was created in a dialog box using the source code and resource files DPTOOLS.CPP and DPTOOLS.RC. Based on the user's selection, double-clicking on session provider instantiated a DirectPlay object.

The next step is to set up the game client-server session. The computer can be the game host (server) or it can be a connecting client. Now take a look at how to make two computers communicate with each other in a client-server environment.

DirectPlay Client or Server?

Once the session provider has been selected and the DirectPlay object instantiated, determine whether the DirectPlay instance is going to be a game client or a game server. For the coding example provided, just use a simple dialog box from the Microsoft Developer Studio dialog editor. The dialog IDD_HOST in DPTOOLS.RC is the default dialog resource provided by the Microsoft Developer Studio dialog editor. I moved the OK push button to the lower-left corner and renamed the caption and resource ID. Double-clicking on the push button in the editor will bring up a push button property dialog. This is how I changed the values of push buttons IDOK and IDCANCEL, provided by the resource editor, to ID_HOST and ID_JOIN. Figure 10-13 shows the resource labels on the Host or Join dialog box.

The driver code for the dialog box is the CALLBACK function DlgProcHost(). This will display the dialog to the user (*Reboot* cartoon terminology for a game player). The dialog will be displayed after the sessions provider dialog designed previously. The user will have the option of double-clicking push button IDC_HOST or IDC_JOIN. If the user selects the button labelled HOST, initialize all DirectPlay and game settings for the node (network term for a computing device on network) to be a DirectPlay game server. If the user selects the push button labelled JOIN, the node will be initialized as a DirectPlay client. The code for DlgProcHost is shown in Listing 10-5.

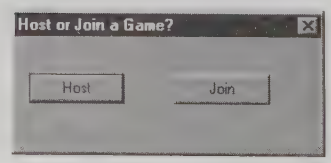


FIGURE 10-13
Dialog IDD_HOST for choosing client or server instantiation

LISTING 10-5 Dialog to host or join a game session

```
#define GAME_HOST   1   // game server
#define GAME_JOIN   2   // game client

BOOL CALLBACK DlgProcHost( HWND hDlg,
                           UINT msg,
                           WPARAM wParam,
                           LPARAM lParam )
{
  switch(msg)
  {
    case WM_COMMAND:
      switch(wParam)
      {
        case IDC_HOST:
          EndDialog(hDlg, GAME_HOST);
          return TRUE;

        case IDC_JOIN:
          EndDialog(hDlg, GAME_JOIN);
          return TRUE;
      }
      break;
  }
  return FALSE;
}
```

Use the Windows API call EndDialog() to return the user selection. Using the switch statement, if the user selects IDC_JOIN, the CALLBACK function returns GAME_JOIN to the calling program. If IDC_HOST is selected, the value GAME_HOST is returned to the calling program. The function GetGameHost() in source file DPTOOLS.CPP is the calling routine for CALLBACK DlgProcHost(). This code is shown in Listing 10-6.

LISTING 10-6 DlgProcHost driver function GetGameHost()

```
int GetGameHost(HWND hwnd)
{
       return (DialogBox(NULL, (LPCSTR) IDD_HOST, hwnd, (DLGPROC)
       ➥DlgProcHost));
}
```

With the DirectPlay node identified as GAME_HOST or GAME_JOIN, now proceed with the instantiation of the DirectPlay object. There is some information that must be provided to DirectPlay before proceeding. Give DirectPlay a Globally Unique Identifier (GUID), the name of the game to play over DirectPlay, and game names. Every game you develop for DirectPlay must have its own GUID. These are really easy to generate. If your environment variables are set up properly for your Win32 SDK, and you have a network card, you can just shell out to a DOS console window and execute,

```
UUIDGEN.EXE
```

The program will return something that looks like,

```
53ebc3d0-2137-11cf-a4f4-00aa0060f461
```

Incorporate this value into your game header file. For the DPTOOLS sample driver, I made a header file called XGAME.H, see Listing 10-7.

LISTING 10-7 Defining a game GUID from the header file XGAME.H

```
#ifndef XGAME_H_
#define XGAME_H_

#define INITGUID
/* 53ebc3d0-2137-11cf-a4f4-00aa0060f461 */
GUID XGAME_GUID = {
       0x53ebc3d0,0x2137,0x11cf,
       {0xa4, 0xf4, 0x00, 0xaa, 0x00, 0x60, 0xf4, 0x61}
};
#endif
```

Just be sure to include a file like this in a manner that the DPTOOLS code can access. Since every DirectPlay game needs to provide a GUID, a game name, and fields for players' names, it made sense to me to provide a game definition data structure. This would provide an extra layer of information hiding for DPTOOLS when using it as a generic DirectPlay game launcher. The GameSessionsDefs data structure declaration is shown in Listing 10-8.

LISTING 10-8 Defining the GameSessionsDefs structure

```
typedef struct
{
  GUID   gameGUID; // Game GUID
  char   szGameName[DPSESSIONNAMELEN]; // Game name
  char   szPlayerAlias[80];
  char   szPlayerFormal[80];
}GameSessionDefs;

GameSessionDefs *gsdfPtr;
```

The fields of this data structure can be initialized with the DPTOOLS routine InitGameDef(). The code to initialize the GameSessionDefs is shown in Listing 10-9.

LISTING 10-9 Initializing the values in the GameSessionDefs structure

```
BOOL InitGameDef(GUID pGuid,
                                 LPSTR szGameName,
                                 LPSTR szPlayerAlias,
                                 LPSTR szPlayerFormal)
{
```

```
  gsdfPtr = new GameSessionDefs;

  if (gsdfPtr == NULL)
    return FALSE;

  gsdfPtr->gameGUID = pGuid;
  strcpy(gsdfPtr->szGameName, szGameName);

  // copy in player alias
  if (szPlayerAlias != NULL)
    strcpy(gsdfPtr->szPlayerFormal, szPlayerFormal);
  else
    memset(gsdfPtr->szPlayerFormal, 0x00, 80);

  // copy in player name
  if (szPlayerAlias != NULL)
    strcpy(gsdfPtr->szPlayerAlias, szPlayerAlias);
  else
    memset(gsdfPtr->szPlayerAlias, 0x00, 80);

  return TRUE;
}
```

This routine is used by both a DirectPlay client and a DirectPlay server. The contents of these fields will be used when DirectPlay opens a session or creates a player. You'll see this in a moment.

With the GUID, DirectPlay session game name, and player information in GameSessionDefs, proceed with the DirectPlay object instantiation, see Listing 10-10.

LISTING 10-10 Initializing the GameSessionDefs structure for client and server

```
int nodeType;  // nodeType HOST or JOIN

nodeType = GetGameHost(hwnd);

switch(nodeType)
{
  case GAME_HOST:
    MessageBox(hwnd,"GAME_HOST","OK",MB_OK);
    bGameServer = TRUE;
    InitGameDef(XGAME_GUID,"XGame", "Server", "XGame Server");
    if (InitDPServer() == FALSE)
      return FALSE;
    break;

  case GAME_JOIN:
    MessageBox(hwnd,"GAME_JOIN","OK",MB_OK);
    bGameServer = FALSE;
```

continued on next page

continued from previous page

```
    InitGameDef(XGAME_GUID,"XGame", "Client", "XGame Client");
    JoinWhichGame(hwnd);
    break;

  default:
    return FALSE;
}
```

Using the switch statement, examine the value *nodeType* which was returned from GetGameHost(). Based on the user selection of the *Host or Join Game?* dialog, set up the session as a client or a server. Notice that the DPTOOLS routine InitGameDefs() is called in both case statements. If the user selected HOST from the dialog, fill in the GameSessionDefs fields *szPlayerAlias* with "Server" and *szPlayerFormal* with "XGame Server". These fields will be valuable when breaking out the network sniffer and examining IPX and UDP packets. Using the sniffer to decode the packets, shows the "XGame Server" and other byte fields. For those of you on tight budgets, I will provide some packet examples. If you're a professional, I strongly recommend that you convince your manager to purchase some form of network analyzer. The Network General sniffers are pretty much the flag ship packet decoders of the industry and are extremely expensive. I am using my Macintosh running an application called NetMinder to decode packets. This is just using the Mac's Farallon NIC to tap into the link layer and see what packets are flying around. If you have a spare PC lying around, as most of us developers do, run Linux. It's a shareware UNIX operating system and has a ton of network tools.



Initializing a DirectPlay Server

Use the Boolean flag *bGameServer*, which is set during the switch statement, to determine client or server functionality. This flag needs to be globally accessible to game application as well. There are examples of why this needed to be done in the discussion on messaging. Listing 10-11 shows the code used for initializing the DirectPlay object as a game server.

LISTING 10-11 Initializing the GameSessionDef structure for a game server

```
case GAME_HOST:
  MessageBox(hwnd,"GAME_HOST","OK",MB_OK);
  bGameServer = TRUE;
  InitGameDef(XGAME_GUID,"XGame", "Server", "XGame Server");
  if (InitDPServer() == FALSE)
    return FALSE;
  break;
```

The flag is set to TRUE if the user selected HOST from the *Host or Join Game?* dialog. For the DPTOOLS example, initialize the fields of the structure *GameSessionDefs* with a call to the routine InitGameDefs(). After the fields are initialized, call the DPTOOLS function InitDPServer(). With InitDPServer(), take the fields of *GameSessionDefs* and initialize a DirectPlay DPSESSIONDESC data structure. The DPSESSIONDESC struct fields are used to describe the capabilities of the DirectPlay session. The fields include how many players the game can allow *(dwMaxPlayers)*, the GUID, and the name of the game. These fields for describing session capabilities are shown in Figure 10-14. The code for InitDPServer() is provided in Listing 10-12.

LISTING 10-12 Initializing the DirectPlay game server

```
BOOL InitDPServer()
{
  HRESULT       dpval;
  DPSESSIONDESC dpsd;

  memset(&dpsd, 0x00, sizeof(DPSESSIONDESC));
  dpsd.dwSize = sizeof(dpsd);
  dpsd.dwMaxPlayers = MAX_PLAYERS;
  dpsd.dwFlags = DPOPEN_CREATESESSION;
  dpsd.guidSession = gsdfPtr->gameGUID;
  strcpy(dpsd.szPassword, "");
  strcpy(dpsd.szSessionName,gsdfPtr->szGameName);

  dpval = lpDP->Open(&dpsd);
  if (dpval != DP_OK)
  {
    lpDP->Release();
    lpDP = NULL;
    MessageBox(GetActiveWindow(),"Open FAILED", "ERROR",MB_OK);
    return FALSE;
  }
  return TRUE;
}
```

The DPSESSIONDESC struct *dpsd*, in the code above, initializes the *dwFlags* field using the DPSESSIONDESC flag DPOPEN_CREATESESSION. Here you are telling DirectPlay, based on the session provider the user selected and the fields provided for the session capabilities, to create a new DirectPlay session. The fields of DPSESSIONDESC are then passed to the DirectPlay member function Open(), where the DirectPlay interface establishes the communication over the session provider selected by the user. The DirectPlay member Open() is shown in Figure 10-15.

If you're creating a DirectPlay session over IP or IPX, you won't see any server activity on your network sniffer. Remember the discussion about socket interfaces in a client-server architecture—the server is a *passive* socket? It sits

```
typedef struct {
        DWORD       dwSize;
        GUID        guidSession;
        DWORD       dwSession;
        DWORD       dwMaxPlayers;
        DWORD       dwCurrentPlayers;
        DWORD       dwFlags;
        char        szSessionName [DPSESSIONNAMELEN];
        char        szUserField [DPUSERRESERVED]'
        DWORD       dwReserved1;
        char        szPassword [DPPASSWORDLEN];
        DWORD       dwReserved2;
        DWORD       dwUser1;
        DWORD       dwUser2;
        DWORD       dwUser3;
        DWORD       dwUser4;
} DPSESSIONDESC;
```

FIGURE 10-14
Describing capabilities of a DirectPlay session

```
HRESULT   Open (
          LPDPSESSIONDESC  lpDPSDESC)

Common Return Values:
          DP_OK
          DPERR_GENERIC
          DPERR_INVALIDOBJECT
          DPERR_USERCANCEL
```

FIGURE 10-15
Opening a communication link over a service provider

and waits for a connection. You really won't see any network activity until you start working with the client code.

Initializing a DirectPlay Client

In this programming example, a DirectPlay client session is initialized if the user selected the JOIN option from the *Host or Join Game?* dialog. The global Boolean flag *bGameServer* is set to FALSE and the DPTOOLS function InitGameDef()

initializes the *GameSessionDef* data structure fields to "XGame Client". As mentioned earlier, I am setting these fields for packet analysis on the network. The case statement to handle the client instantiation of the DirectPlay object is shown in Listing 10-13.

LISTING 10-13 Initializing the GameSessionDefs structure for a game client

```
case GAME_JOIN:
  MessageBox(hwnd,"GAME_JOIN","OK",MB_OK);
  bGameServer = FALSE;
  InitGameDef(XGAME_GUID,"XGame", "Client", "XGame Client");
  JoinWhichGame(hwnd);
  break;
```

The case statement looks similar to the way GAME_HOST (a server session) was handled, except the fields were initialized for a DirectPlay client session. The obvious difference in this code is that you don't call the DPTOOLS function InitDPServer(). Instead, use the DPTOOLS function JoinWhichGame(). This code is going to be slightly more complicated than what was used for InitDPServer().

As a DirectPlay session client you're going to have to establish a communication link to the DirectPlay session server. Using the client server architecture and TCP/IP as an example, the client is going to use the active socket interface. The client node is going to go out onto the network and request a server. If a server responds, an exchange of game session data will take place between the two nodes. This exchange will be everything from low-level hardware addresses to game-specific parameters. Some game information will require interaction from the user on the client node. This means data will be pulled down from the game server and presenting in Windows dialogs and list boxes. The type of information pulled down from the server is descriptive strings and GUIDs of games available. The DPTOOLS routine used to handle this seemingly complicated operation is CALLBACK DlgProcWhichGame(). This code for joining a game session is presented in Listing 10-14.

LISTING 10-14 Joining an active game session

```
BOOL CALLBACK DlgProcWhichGame( HWND hDlg,
                                UINT msg,
                                WPARAM wParam,
                                LPARAM lParam)

{
  static LONG    iIndex;
  static HWND    hWndCtl;
  DPSESSIONDESC dpsd;
  HRESULT        dpval;
```

continued on next page

continued from previous page

```
switch(msg)
{
  case WM_INITDIALOG:
    hWndCtl = GetDlgItem(hDlg, IDC_LIST_WHICHGAME);
    if (hWndCtl == NULL)
    {
      EndDialog(hDlg, TRUE);
      return TRUE;
    }
    memset(&dpsd, 0x00, sizeof(DPSESSIONDESC));
    dpsd.dwSize = sizeof(dpsd);
    dpsd.guidSession = gsdfPtr->gameGUID;

    lpDP->EnumSessions(&dpsd,(DWORD) 5000, EnumGames,
                       (LPVOID) hWndCtl, (DWORD)NULL);
    SetFocus(hWndCtl);
    return FALSE;

  case WM_COMMAND:
    switch(HIWORD(wParam))
    {
      case LBN_SELCHANGE:
        iIndex = SendMessage((HWND) lParam, LB_GETCURSEL, 0, 0);
        hWndCtl = (HWND) lParam;
        return FALSE;

      case 0:
        if (LOWORD(wParam) == IDCANCEL)
        {
          lpDP->Close();
          lpDP->Release();
          lpDP = NULL;
          EndDialog(hDlg, FALSE);
          return TRUE;
        }

      case LBN_DBLCLK:
        if (HIWORD(wParam) == LBN_DBLCLK)
        {
          hWndCtl = (HWND) lParam;
          iIndex = SendMessage( hWndCtl, LB_GETCURSEL, 0, 0);
        }
       if (iIndex != LB_ERR)
       {
         memset(&dpsd, 0x00, sizeof(DPSESSIONDESC));
         dpsd.dwSize = sizeof(dpsd);
         dpsd.dwFlags = DPOPEN_OPENSESSION;
         dpsd.guidSession =    gsdfPtr->gameGUID;
         dpsd.dwSession = SendMessage((HWND) hWndCtl,
                          LB_GETITEMDATA, iIndex, 0);
```

```
            dpval = lpDP->Open(&dpsd);
            if (dpval != DP_OK)
            {
              lpDP->Close();
              lpDP->Release();
              lpDP = NULL;
              EndDialog(hDlg, FALSE);
            }
            EndDialog(hDlg, TRUE);
            return TRUE;
          } // if !LB_ERR
        } // switch
      } // switch msg
      return FALSE;
}
```

Rather than looking at this code as being overwhelming, take the advice of the psychiatrist, Dr. Leo Marvin, from the movie, *What About Bob?*. Baby steps to the elevator, baby steps onto the bus, and baby steps through the dialog code. Provide the user with a dialog box that contains an enumerated listing of game sessions identified by a particular GUID. The user will double-click on the session to join in the game.

Now take baby steps to fill in the dialog list box. Generate an enumerated list of game sessions in progress before displaying the dialog. The Windows message WM_INITDIALOG triggers DirectPlay to generate an enumerated list box of game sessions in progress with the GUID you are requesting. This snippet of code is shown in Listing 10-15.

LISTING 10-15 Initializing the EnumSessions list box

```
case WM_INITDIALOG:
  hWndCtl = GetDlgItem(hDlg, IDC_LIST_WHICHGAME);
  if (hWndCtl == NULL)
  {
    EndDialog(hDlg, TRUE);
    return TRUE;
  }
  memset(&dpsd, 0x00, sizeof(DPSESSIONDESC));
  dpsd.dwSize = sizeof(dpsd);
  dpsd.guidSession = gsdfPtr->gameGUID;

  lpDP->EnumSessions(&dpsd,(DWORD) 5000, EnumGames,
                          (LPVOID) hWndCtl, (DWORD)NULL);
  SetFocus(hWndCtl);
  return FALSE;
```

The dialog box used in this example is in DPTOOLS.RC; its resource ID is IDD_WHICHGAME. If the Microsoft Developer Studio isn't fired up or handy at the moment, the dialog box is shown in Figure 10-16.

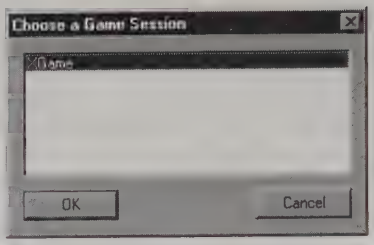


FIGURE 10-16
Dialog Box IDD_WHICHGAME to enumerate game sessions

When Windows dispatches the message WM_INITDIALOG, add the enumerated list to the dialog list box using the DirectPlay member EnumSessions. The DirectPlay member also requires providing your own CALLBACK function to be called for each DirectPlay session that responds. The prototype for the CALLBACK function is shown in Figure 10-17 along with the DirectPlay member EnumSessions that calls it.

The CALLBACK function that must be provided for the DirectPlay member EnumSessions() is shown in Listing 10-16.

LISTING 10-16 The *Spells of Fury* callback function EnumGames()

```
BOOL FAR PASCAL EnumGames(
                          LPDPSESSIONDESC lpDPGameDesc,
                          LPVOID lpContext,
                          LPDWORD lpdwTimeOut,
                          DWORD dwFlags )
{
  LONG iIndex;
  HWND hWnd = (HWND) lpContext;

  if (dwFlags & DPESC_TIMEDOUT)
    return FALSE;

  iIndex = SendMessage(hWnd, LB_ADDSTRING, 0,
             (LPARAM)lpDPGameDesc->szSessionName);
  if (iIndex != LB_ERR)
    SendMessage(hWnd, LB_SETITEMDATA,
             iIndex, (LPARAM)lpDPGameDesc->dwSession);
  SetFocus(hWnd);
  SendMessage(hWnd, LB_SETCURSEL, 0, 0);
  return TRUE;
}
```

The function essentially retrieves the game DPSESSIONDESC from each response and adds the name of the game available to the list box. After adding the string, it returns to the calling DirectPlay member EnumSessions().

Now take a look at network IP traffic when DirectPlay member EnumSessions() polls servers for games in progress. Figure 10-18 shows an example of a simple test network. This network has a Sun workstation with an example IP address of 128.146.50.8; a DirectPlay client PC, with IP address

```
                        EnumSessions (
HRESULT                 LPDIRECTPLAY                lpDirectPlay,
                        LPDPSESSIONDESC             lpSDesc,
                        DWORD                       dwTimeOut
                        LPDPENUMSESSIONSCALLBACK    lpEnumCALLBACK,
                        LPVOID                      lpvContext,
                        DWORD                       dwFlags)

Common Return Values:

                        DP_OK
                        DPERR_INVALIDOBJECT
                        DPERR_INVALIDPARAMS

LPDPENUMSESSIONCALLBACK:

                        BOOL  Enum CALLBACK (
                              LPDPSESSIONDESC       lpDPSGame Desc,
                              LPVOID                lpContext,
                              LPDWORD               lpdwTimeOut,
                              DWORD                 dwFlags)
```

FIGURE 10-17
Getting an enumerated listing of active games

128.146.50.127; and a DirectPlay server PC, whose IP address is 128.146.50.22. I originally wrote the DPTOOLS driver to use with a network analyzer. You'll notice as you execute the code that I used MessageBox breakpoints. I left these in so you could experiment with IP or IPX on your network sniffer. For this example I'll provide an explanation for what's taking place over a WinSock IP session.

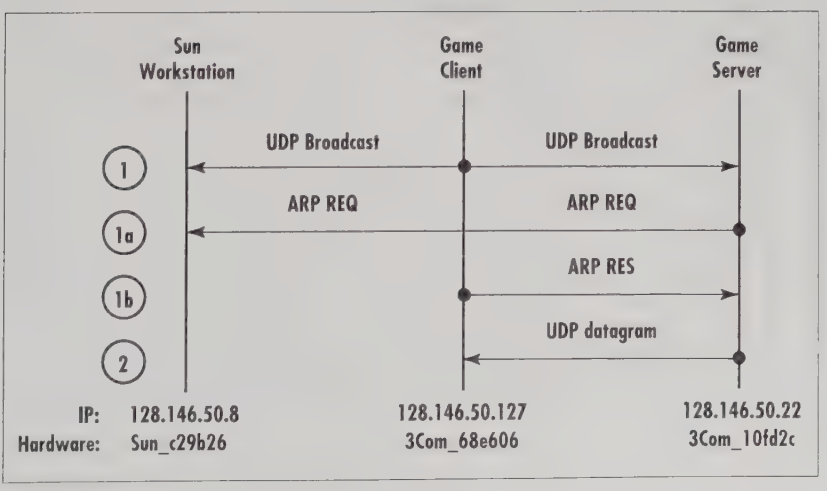


FIGURE 10-18
IP traffic on a network, based on EnumSessions()

As was mentioned a few times earlier, the server socket is passive. The server sits there on the net like the Maytag service man waiting for a ring from a client. When the InitDPServer() is executed, there is no activity on the net. However, on the client, when Windows dispatches the message WM_INITDIALOG, it triggers the DirectPlay member EnumSessions() and you begin to see activity on the net. A UDP datagram is broadcast (destination address is 255.255.255.255), from the client, to request a server with a specific port ID. In Figure 10-18, the UDP datagram is sent to all nodes on the segment. The Sun workstation sees the datagram but does not respond because it knows nothing about the port ID the DirectPlay client is requesting. The next steps that are labeled 1b and 1c in Figure 10-18 are ARPs sent out by the server and the client to exchange hardware addresses. On my network, I only have four devices. In this example, the two PCs haven't communicated over TCP/IP to each other yet, so they don't have IP to hardware address mappings in their ARP cache. If you're on a corporate net, you may not see this ARP occur because the mappings may already exist in each device's ARP cache.

What is taking place with the ARP REQ(uest) in event 1b is that the DirectPlay session server reads the UDP datagram and understands the message. It puts an ARP REQ on the net for IP 128.146.50.127, the DirectPlay client, to send its hardware address. In event 1c, in Figure 10-18, the ARP RES(ponse) is sent from the DirectPlay client that contains its 48-bit hardware address. With the information stored in ARP cache on both devices, a UDP datagram is sent from the DirectPlay session server with descriptive information about the game session it is servicing.

The sniffer provides information from the Ethernet frame all the way to the UDP datagram.You only need to care about the UDP datagram and its contents. The following datagram is the response sent by the server to the client for what game sessions it handles.

```
UDP Header
Src. Port          2224
Dest. Port         1025
Length             158
Checksum          $744c

Packet Data
4b 79 72 61 48 61 6c 6c 19 01 00 00 02 00 00 00   KyraHall........
7c 00 00 00 d0 c3 eb 53 37 21 cf 11 61 00 00 00    |......S7!..a...
00 00 00 00 00 00 00 00 10 00 00 00 01 00 00 00   ................
02 00 00 00 58 47 61 6d 65 00 00 00 00 00 00 00   ....XGame.......
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00  00 00 00 00 02 00 00 00  ................
b0 08 01 00 01 00           ......
```

You'll notice in the datagram that the server sends the game session name XGAME. One thing I found of interest is that the first 8 bytes of a DirectPlay

datagram always contain the string KyraHall. (According to the *DOOM95* developers, Kyra Hall is the name of the original DirectPlay programmer's daughter.) Typical traffic from DirectPlay member EnumSessions() will include a UDP datagram from a client requesting a server port ID . If a server recognizes the port ID, it will respond with a UDP datagram advertising the service. If the two devices don't have IP to hardware mappings in their ARP cache, the server will send an ARP REQ and the client will send an ARP RES.

This is basically all the code does when Windows 95 dispatches the WM_INITDIALOG message. You request an enumerated listing of game sessions that is identified by the game GUID. This action will result in at least two UDP datagrams on the network, assuming there is only one client and one server on a lab network. There could be multiple servers on the same segment running a copy of the XGAME example with the same GUID. The users will have to select which session they wish to join.

Now, take baby steps to the WM_COMMAND message of the dialog procedure which processes the user's selection. When this section of code executes, the user will be seeing the list box of XGAME sessions available. Wait for the user to select a session by double-clicking on a list box field. Listing 10-17 shows how to request an enumerated list of game sessions active.

LISTING 10-17 Requesting a list of game messages

```
case WM_COMMAND:
  switch(HIWORD(wParam))
  {
    case LBN_SELCHANGE:
      iIndex = SendMessage((HWND) lParam, LB_GETCURSEL, 0, 0);
      hWndCtl = (HWND) lParam;
      return FALSE;

    case 0:
      if (LOWORD(wParam) == IDCANCEL)
      {
        lpDP->Close();
        lpDP->Release();
        lpDP = NULL;
        EndDialog(hDlg, FALSE);
        return TRUE;
      }

   case LBN_DBLCLK:
     if (HIWORD(wParam) == LBN_DBLCLK)
     {
       hWndCtl = (HWND) lParam;
       iIndex = SendMessage( hWndCtl, LB_GETCURSEL, 0, 0);
     }
     if (iIndex != LB_ERR)
     {
```

continued on next page

continued from previous page

```
        memset(&dpsd, 0x00, sizeof(DPSESSIONDESC));
        dpsd.dwSize = sizeof(dpsd);
        dpsd.dwFlags = DPOPEN_OPENSESSION;
        dpsd.guidSession = gsdfPtr->gameGUID;
        dpsd.dwSession = SendMessage((HWND) hWndCtl,
                         LB_GETITEMDATA, iIndex, 0);
        dpval = lpDP->Open(&dpsd);
        if (dpval != DP_OK)
        {
          lpDP->Close();
          lpDP->Release();
          lpDP = NULL;
          EndDialog(hDlg, FALSE);
        }
        EndDialog(hDlg, TRUE);
        return TRUE;
      } // if !LB_ERR
    } // switch
```

When the users see the list box of game sessions in progress they can choose a selection by double-clicking on the desired session. They may also choose to push the cancel button and not play a game. The simplest case is if a user chooses not to play a game and selects the cancel button. This is handled by the case statement case 0:. If the user chooses not to play, shut down DirectPlay.

Now what's left is handling the case statement where the user has double-clicked on a game session. If the user selects a game session you must initialize the DPSESSIONDESC structure for the session you wish to make a connection to. This code is shown in Listing 10-18.

LISTING 10-18 Initializing the DPSESSIONDESC to connect to a session

```
memset(&dpsd, 0x00, sizeof(DPSESSIONDESC));
dpsd.dwSize = sizeof(dpsd);
dpsd.dwFlags = DPOPEN_OPENSESSION;
dpsd.guidSession = gsdfPtr->gameGUID;
dpsd.dwSession = SendMessage((HWND)hWndCtl,LB_GETITEMDATA,iIndex,0);
```

This looks similar to what was done to initialize the server DPSESSIONDESC. The primary difference is setting up a DirectPlay instance to connect to a session rather than creating one. Inform DirectPlay to connect to a session by assigning the *dwFlags* field with the flag DPOPEN_OPENSESSION. Also inform DirectPlay what session you wish to connect to. From the GameSessionDef struct, grab the game GUID and hand it off to DirectPlay in the DPSESSIONDESC field *guidSession*. Note that multiple sessions of XGAME can be running on the same net. Leave it up to the user to select which session. When they double-click on the list box field, this information can be retrieved with the Windows message

LB_GETITEMDATA. This is the information assigned to the *dwSession* field. You now have enough information to connect to the game. This connection is made by a call to the DirectPlay member Open(). For more information on this DirectPlay interface, see Figure 10-15 on opening a communication link. The code for DirectPlay to open a connection is shown in Listing 10-19.

LISTING 10-19 Opening a DirectPlay session

```
MessageBox( GetActiveWindow(),"Sniffer Pause","DEBUG lpDP->Open Start", MB_OK);
dpval = lpDP->Open(&dpsd);
if (dpval != DP_OK)
{
  lpDP->Close();
  lpDP->Release();
  lpDP = NULL;
  EndDialog(hDlg, FALSE);
}
MessageBox( GetActiveWindow(),"Sniffer Pause","DEBUG lpDP->Open Done", MB_OK);
EndDialog(hDlg, TRUE);
return TRUE;
```

I have placed sniffer breakpoints in this code as well. This will give you a chance to pause and get your network analyzer ready to capture data, or just break as you step through the code that is executing. If you don't have a network analyzer, don't worry. I have provided all the information from my analysis of the datagrams. The breakpoints coincide with the datagrams listed in this chapter. The sample network traffic from the Fisher-Price network is shown in Figure 10-19. When the client calls the DirectPlay member Open(), DirectPlay puts a datagram on the net to request game session information. The server providing the XGAME session selected returns datagrams with information about the session. This is a DPSESSIONDESC synchronization taking place. The connection is being established and the server is sending its session data. In response to the datagram sent by the client, the server responds with two datagrams. The first datagram advertises its *szSessionName*, "XGAME". The second datagram advertises the sessions *szPlayerFormalName*, "XGame Server" and its *szPlayerAlias*, "Server". The latter datagram's decoding looks like,

```
UDP Header
Src. Port          2308
Dest. Port         2308
Length             128
Checksum           $2f5b

Packet Data
4b 48 30 55 01 00 00 00 1d 01 6c 00 0f 00 00 00    KHOU.....l.....
01 00 00 00 01 00 00 00 58 47 61 6d 65 20 53 65    ........XGame Se
72 76 65 72 00 b5 2e 7e 02 00 00 00 0d 01 00 00    rver...~........
d4 f4 a9 00 28 00 00 00 d4 f4 a9 00 28 00 00 00      ....(.......(...
98 f4 a9 00 00 00 00 00 94 f7 a9 00 53 65 72 76    ............Serv
```

```
65 72 00 7e 02 00 00 00 0d 01 00 00 d4 f4 a9 00     er.~............
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
00 00 00 00 84 f6 a9 00     ........
```

When the DirectPlay member Open() finishes executing, the dialog box terminates and a Boolean value is returned to the calling program. If the call is successful, a communication link is established between the client and the server. The DPTOOLS CALLBACK function DlgProcWhichGame() is fairly complicated in what it does. But, again, if you follow Dr. Leo Marvin's advice and take *baby steps* through it, you can piece together what is taking place when a client connects. The DPTOOLS function JoinWhichGame() launches the CALLBACK function DlgProcWhichGame(). Listing 10-20 provides the code for joining a game session.

LISTING 10-20 Dialog box to join an active DirectPlay session

```
int JoinWhichGame(HWND hwnd)
{
  return(DialogBox(NULL,
              (LPSTR) IDD_WHICHGAME,
              hwnd, (DLGPROC) DlgProcWhichGame));
}
```

You now have the code to open a client session and create a server session. Pull the code together now and write a driver for the DirectPlay game launcher. Next take a look at how to pull this all together to create a DirectPlay driver for the generic game application constructed back in Chapter 1.



A Generic Client-Server Driver for XGame

When the DirectPlay-capable Windows 95 generic game application executes, you don't know whether the user intends to be a game client or server. The possibilities for either case must exist at runtime. Leave the decision up to the user to determine on how the DirectPlay session will be instantiated. This

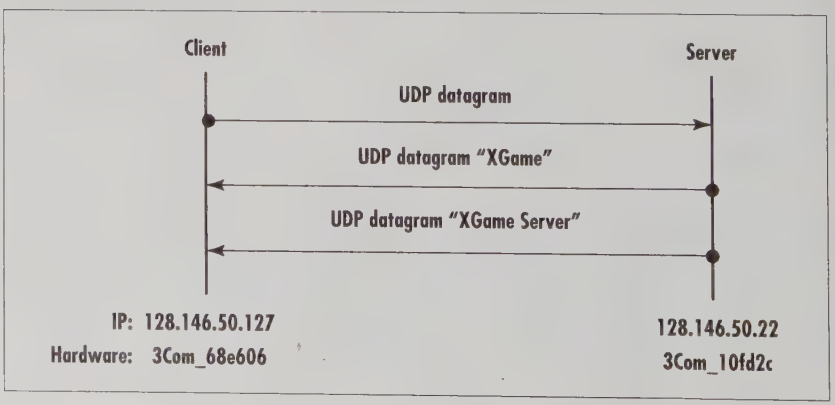


FIGURE 10-19
IP network activity generated by client's Open() call

means that you must make provisions in the code to handle both a DirectPlay client or server object instantiation. All the code already exists to create the server or for the client to connect. You need to develop a driver to handle a little of the decision-making, based upon the user's input. Putting together all the code that's been developed produces the DPTOOLS function SessionDriver(); see Listing 10-21.

LISTING 10-21 The *Spells of Fury* DPTOOLS function SessionDriver()

```
BOOL SessionDriver(HWND hwnd)
{
  int nodeType;  // nodeType HOST or JOIN
  HRESULT dpval;

  if (lpDP != NULL)
  return FALSE;

  // Get Session Provider (Modem, IPX, IP)
  GetProvider(hwnd);

  if (lpDP == NULL)
  {
    MessageBox(hwnd, "GetProvider FAILED", "ERROR", MB_OK);
    return FALSE;
  }

  // This node game HOST or JOIN
  nodeType = GetGameHost(hwnd);

  switch(nodeType)
  {
    case GAME_HOST:
      MessageBox(hwnd,"GAME_HOST","OK",MB_OK);
      bGameServer = TRUE;
      InitGameDef(XGAME_GUID,"XGame", "Server", "XGame Server");
      if (InitDPServer() == FALSE)
        return FALSE;
      break;

    case GAME_JOIN:
      MessageBox(hwnd,"GAME_JOIN","OK",MB_OK);
      bGameServer = FALSE;
      InitGameDef(XGAME_GUID,"XGame", "Client", "XGame Client");
      JoinWhichGame(hwnd);
      break;

    default:
      return FALSE;
  }
  dpval = lpDP->CreatePlayer(&dpID, gsdfPtr->szPlayerAlias,
  gsdfPtr->szPlayerFormal, &hDPEvent);
```

continued on next page

continued from previous page

```
    if(dpval != DP_OK)
    {
      lpDP->Close();
      lpDP->Release();
      lpDP = NULL;
      return FALSE;
    }
    return TRUE;
}
```

All of this code should look familiar by now. The DPTOOLS function GetProvider() generates the list box of what DirectPlay session service providers are available. The DPTOOLS function GetGameHost() launches a dialog to ask the users if they wish to host (be a game server) or join (connect as a client) a game session. The switch statement should look familiar too; these conditions were used to create the server or open the client session. The only new code in this function is the call to DirectPlay member CreatePlayer(). The interface to CreatePlayer() is shown in Figure 10-20.

CreatePlayer() is used to identify new players when they are joining a game. Every node in the DirectPlay session calls CreatePlayer(). This is used to keep track of who is joining a game. Both the client and server nodes call CreatePlayer(). The nodes will furnish information on the player's formal name and alias. In return, DirectPlay gives the node a DirectPlay ID. This ID is used to keep track of what nodes are in the session and when DirectPlay messages are sent to remote players.

Figure 10-21 shows a sample of network activity on a lab network where the DirectPlay client calls CreatePlayer(). The client sends a datagram to the

```
HRESULT  CreatePlayer (
         LPDPID         lpDPId,
         LPSTR          lpPlayerNameAlias,
         LPSTR          lpPlayerNameFormal
         LPHANDLE       lpPlayerHandle)

Common Return Values:

         DP_OK
         DPERR_CANTCREATEPLAYER
         DPERR_INVALIDOBJECT
         DPERR_INVALIDPARAM
```

FIGURE 10-20

Creating a DirectPlay game player

server advertising the player's formal name and alias. The server responds to the client with a datagram that has the client's registered formal name, alias, and assigned DirectPlay DPID. The packet decoding of the server's response looks like,

```
UDP Header
Src. Port           2308
Dest. Port          2308
Length              128
Checksum            $666d

Packet Data
4b 48 00 00 00 00 00 00 1f 01 6c 00 03 00 00 00   KH.......l.....
01 00 00 00 00 00 00 00 58 47 61 6d 65 20 43 6c   ........XGame Cl
69 65 6e 74 00 1f 2f 01 8a 8b 6e 4c 17 01 64 13    ient./...nL..d.
2f 01 00 00 00 00 c0 00 00 00 8a 8b 00 00 7a 8b    /.............z.
63 00 30 1f 00 00 37 1f 00 00 01 00 43 6c 69 65    c.0..7....Clie
6e 74 00 00 2f 01 ff 00 02 02 bf 00 60 43 37 1f    nt../.......'C7
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 02 00 00 00    ........
```

This datagram contains the DPID for the DirectPlay session client assigned by the session server. If you are confused by the different port numbers in these traces, I used different DirectPlay sessions to perform these network captures. Remember, the port numbers are assigned at runtime. If I were to capture traces from start to finish with one DirectPlay session, all the port IDs would match. For instance, if I used this particular instance, my captures for DirectPlay Open(), CreatePlayer(), and EnumSessions(), would all have source and destination port IDs of 2308.

With a DPID assigned to a session, you have successfully joined a DirectPlay game session. You can't really play anything yet. Only the means to join a game have been provided. Now it is necessary to provide a means of interprocess communication, called messaging, to play the game.

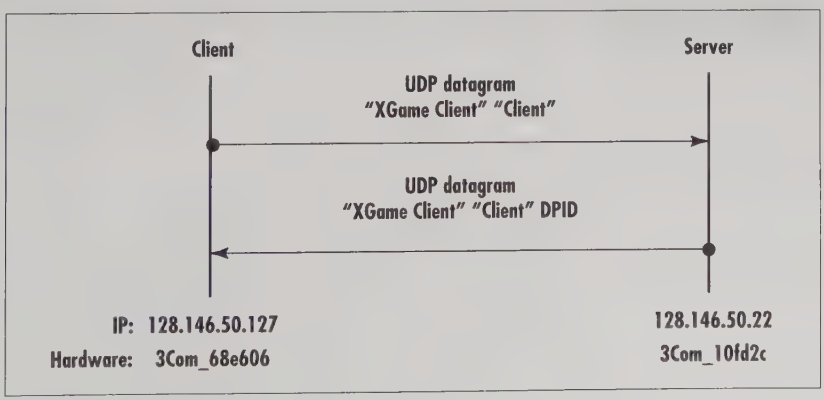


FIGURE 10-21
IP network activity generated by client call CreatePlayer()

DirectPlay Distributed Gaming and Messaging

A connection is needed between two devices to form a DirectPlay session. The DirectPlay sessions are essentially separate processes operating simultaneously that require updates from other processes belonging to the same DirectPlay session. The central process is the session server. This process must communicate with other processes on the wire and send updated information as other processes on the wire (network or modem) provide it. The server must process this information and send it back out to the other remote processes belonging to the session. This makes a distributed computing environment when two external processes share information and update each other.

Sending a Message

A process update is called a message. In DirectPlay, a message has a maximum size of 512 bytes. A message can be really anything you want as long it's no longer than this fixed size limit. For a programming example, make a trivial TCP/IP style connection handshake using DirectPlay messages. Send a network HELLO message. You don't know what service provider the user has selected. You could be running over IP, IPX, or a modem connection. You don't care; you're going to simulate the HELLO using strictly messaging. Constructing a simple message needs at least one byte for the message and a DWORD for the DirectPlayer DPID (the message recipient); see Listing 10-22.

LISTING 10-22 A simple DirectPlay message structure, HELLOMSG

```
typedef struct
{
  BYTE  msg;
  DWORD dpid;
} HELLOMSG, *LPHELLOMSG;
```

We can create some simple one-byte messages to send over the wire. These could include a server HELLO to identify a server, a client HELLO to identify a client, and some HELLO acknowledgment messages to respond to the various HELLOs. Let's think about this—I send from my client node a HELLO message to a server, and the server sends me back an acknowledgment. Where have we seen this before? Sounds like we're writing our own little network protocols. Listing 10-23 shows our home brew list of protocol messages.

LISTING 10-23 Defining simple protocol messages

```
#define MSG_SERVER_HELLO 0x1
#define MSG_SERVER_ACK   0x2
#define MSG_CLIENT_HELLO 0x3
#define MSG_CLIENT_ACK   0x4
```

These messages are arbitrarily defined specifically for the application. The DirectPlay specifications don't have a page defining what a message should look like. If you're new to device drivers and hardware programming, which with messaging is about as close as you can come, you should organize your messages in some fashion. For instance, I tend to reserve messages 0x00 to 0x20, for standard handshake messages. These are the messages you will use from application to application. If you work in a professional environment, you may want your company to standardize on these. That is, all developers in your various groups agree on what the messages from 0x00 to 0x20 are. This leaves messages 0x21 to 0xff for specific product use. The point here is, don't assign these values at random or haphazardly in your header file or library. Make them represent a sequence of events or operations.

We can send our message with the DirectPlay member Send(); this interface is shown in Figure 10-22. We can send a simple message by providing a struct to hold our data. We will use the struct LPHELLOMSG we declared earlier. The code for sending a HELLOMSG is shown in Listing 10-24.

LISTING 10-24 Sending the HELLOMSG using DirectPlay

```
#define MSG_BUFFER_LEN 256
char dpMsgBuf[MSG_BUFFER_LEN];

int msgLen = sizeof(HELLOMSG);
LPHELLOMSG lpHelloMsg;

DWORD pDP->Send( dpID, receiver, 0, (LPSTR) dpMsgBuf, msgLen);

lpHelloMsg = (LPHELLOMSG) dpMsgBuf;
lpHelloMsg->msg = MSG_CLIENT_HELLO;
lpHelloMsg->dpid = dpID;
msgLen = sizeof(HELLOMSG);

pDP->Send( dpID, receiver, 0, (LPSTR) dpMsgBuf, msgLen);
```

This coding example declares a message buffer, *dpMsgBuf*, of 256 bytes, which will be used to pass our message to the DirectPlay member Send(). The struct pointer, *lpHelloMsg*, was assigned to reference the buffer *dpMsgBuf* (typecast LPHELLOMSG). Assign the LPHELLOMSG field *msg*, the one-message MSG_CLIENT_HELLO. The member function, Send(), requires a message length in bytes accompanying the call. Assign the integer variable *msgLen* the size in bytes of the message. The last piece of information the Send() call requires is a destination DPID to deliver the message.

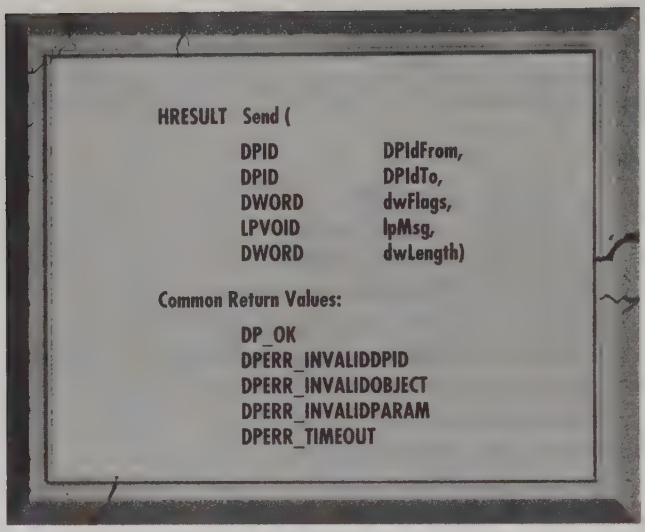


FIGURE 10-22
Sending a DirectPlay message

When you start working with messages, you'll find that you may add all your messages to one header file so that they will be globally accessible. If this is the case, you may find it convenient to write a send function that understands how to send each message. For instance, you could use the code found in Listing 10-25 to handle sending the various hello and acknowledge messages.

LISTING 10-25 Server HELLO message with a client ACK response

```
void DPSend(BYTE msg, DWORD receiver)
{
  int                       msgLen;
  LPHELLOMSG  lpHelloMsg;

  switch (msg)
  {
    case MSG_SERVER_HELLO:
      lpHelloMsg = (LPHELLOMSG) dpMsgBuf;
      lpHelloMsg->msg = msg;
      lpHelloMsg->dpid = dpID;
      msgLen = sizeof(HELLOMSG);
      break;

    case MSG_CLIENT_HELLO:
      lpHelloMsg = (LPHELLOMSG) dpMsgBuf;
      lpHelloMsg->msg = msg;
      lpHelloMsg->dpid = dpID;
      msgLen = sizeof(HELLOMSG);
      break;
  }// switch msg

  lpDP->Send( dpID, receiver, 0, (LPSTR) dpMsgBuf, msgLen);
}
```

This form of sending will work if you understand our messaging and have the messages all predefined somewhere. Encapsulating the send functionality, using only two parameters, BYTE msg and DWORD receiver, can be very convenient. This code provides a means to produce some chatter on the net. Now, take a look at how to read these messages.



Receiving a Message

Receiving a message is a little bit trickier than sending a message. It's not any more difficult than sending; it's just that more attention needs to be paid to the receiving end. A message is arbitrarily sent; there is no idea when or why a message is sent. You must keep one ear to the rail and wait for a message to come in. However, in a message-based environment you can only listen for a finite amount of time and then relinquish time to another process. This is the Windows 3.1 way of doing things. This example will utilize this technique because it could present itself as the worst case as well as the easiest to code. If you pick up on how the message receiving behaves, I recommend coding your message receiving using separate threads.

If you were to code a generic DirectPlay message processor, the code would be that shown in Listing 10-26.

LISTING 10-26 Reading a DirectPlay message

```
#define DPID_BROADCAST   0

void DPRecv()
{
  HRESULT  dpval;
  DWORD    msgLen;
  DPID     dpFrom, dpTo;

  while(1)
  {
    msgLen = MSG_BUFFER_LEN;
    dpval = lpDP->Receive(&dpFrom, &dpTo,
                    DPRECEIVE_ALL, dpMsgBuf,     &msgLen);

    switch(dpval)
    {
      case DP_OK:
        if (dpFrom == DPID_BROADCAST)
        {
          // process system broadcast
        }
        else
        {
```

continued on next page

continued from previous page

```
            // process critical game messages
          }
          break;
        case DPERR_NOMESSAGES:
          return;
        default:
          break;
      } // switch dpval
    } // while(1)
}
```

This is the generic message receiver to use with the Windows 95 generic game application. It's basically an endless loop that calls DirectPlay member Receive(), shown in Figure 10-23. The parameters look similar to the member Send(). Use the dpMsgBuf message buffer to store the message and process game-specific messages externally.

The source code in DPTOOLS uses function GameMsg() to process the game messages. For an example, use the hello and acknowledge messages used earlier as game messages. This function is going to replace the comment // *process critical game messages* in function DPRecv(). The code for GameMsg() is shown in Listing 10-27.

LISTING 10-27 Processing Our Game Messages

```
void GameMsg(int msg, DWORD msgLen)
{
  switch(msg)
  {
    case MSG_SERVER_HELLO:
      break;

    case MSG_CLIENT_HELLO:
      MessageBox(GetActiveWindow(),"CLIENT_HELLO", "GameMsg", MB_OK);
      dpNodeID = ReadHELLOMsg();
      SendACK(MSG_SERVER_ACK, dpNodeID);
      break;

    case MSG_SERVER_ACK:
      MessageBox(GetActiveWindow(),"SERVER_ACK", "GameMsg", MB_OK);
      break;

    case MSG_CLIENT_ACK:
      MessageBox(GetActiveWindow(),"CLIENT_ACK", "GameMsg", MB_OK);
      break;

    default:
      break;
  } // switch msg
}
```

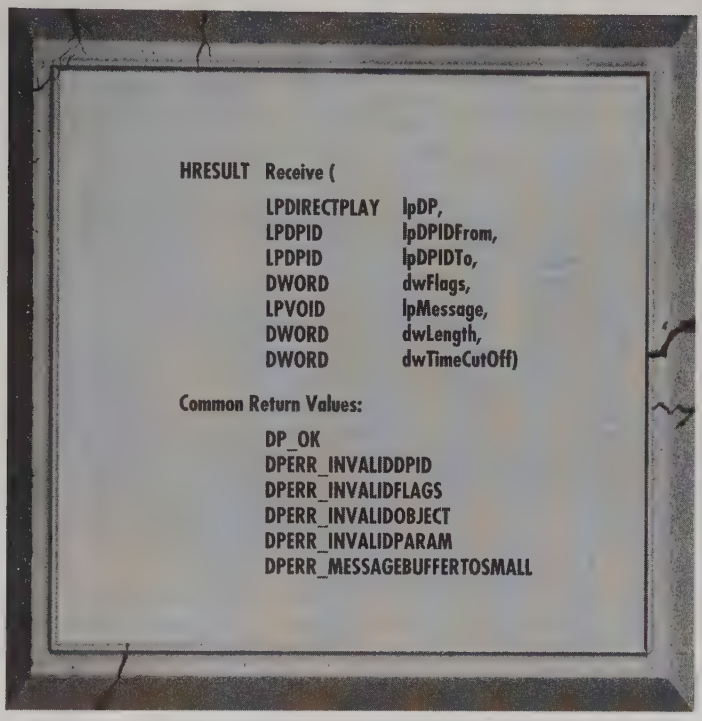


FIGURE 10-23
Receiving a DirectPlay message

The code in this example is kept fairly trivial so you can go in and experiment with it. If the function DPRecv() retrieves a MSG_SERVER_HELLO, it will do nothing. If the message MSG_CLIENT_HELLO is pulled from dpMsgBuf, you read the message and grab the DPID from the message. Use this DPID and send a MSG_SERVER_ACK back to the client. The function to read the MSG_CLIENT_HELLO, ReadHelloMsg(), is shown in Listing 10-28.

LISTING 10-28 Reading the HELLOMSG

```
DPID ReadHELLOMsg()
{
  LPHELLOMSG lpHelloMsg;
  DPID       dpNodeID;
  BYTE       msg;

  lpHelloMsg = (LPHELLOMSG) dpMsgBuf;
  msg = lpHelloMsg->msg;
  dpNodeID = lpHelloMsg->dpid;
  if (bGameServer == TRUE)
    SendACK(MSG_SERVER_ACK, dpNodeID);
  else
    SendACK(MSG_CLIENT_ACK, dpNodeID);
  return (DPID) dpNodeID;
}
```

The big question is how and where do we read in a message in the first place? If we're sticking to the Windows 3.1 methodology, we want to read DirectPlay messages while the Windows message processing is idle. We would call DPRecv(), using the message loop shown in Listing 10-29, from the generic game application, GENX.CPP.

LISTING 10-29 Reading Messages from the Message Loop

```
while(1)
{
  if(PeekMessage(&msg, NULL,0,0,PM_NOREMOVE))
  {
    if(!GetMessage (&msg, NULL, 0, 0))
      return msg.wParam;
    TranslateMessage (&msg);
    DispatchMessage (&msg);
  }
  else
  {
    // do critical game stuff
    DPRecv();
  }
}
return msg.wParam;
```

This style of reading messages is okay for prototyping and simple games, but you'll want to use a more sophisticated approach in your commercial code. This means developing a messaging thread for your application. My approach would be to use a messaging thread running concurrently with the game application. The thread would update the game application as messages came in.



DirectPlay Messaging and a Tournament Server

Use a Direct X application called NetChess—which communicates over a DirectPlay connection—to demonstrate DirectPlay messaging. The messaging used by NetChess shows one solution to a simple problem involving a tournament style game where each player makes a move when the other has finished. NetChess is a simple two player game that requires the server to keep track of whose turn it is and to notify the player. The client has no method of determining whose move it is other than it receives a message from the server that allows it to take a turn.

Originally, when NetChess was developed, it was designed as a two-player chess game on a single computer. When one player completed his move, he

would relinquish the mouse to the opposing player to make a move. Player One moved the white pieces on the board, while Player Two moved the black pieces. Player One and Player Two, in the original chess game, are defined in the following manner,

```
#define WHITE   1
#define BLACK   0
```

When the chess game initializes, Player One, denoted by the label WHITE, is always the first to move. An integer variable *sideMove* keeps track of whose move it is in the game. When Player One successfully moves a piece, the variable *sideMove* is updated. This looks like,

```
sideMove = (sideMove + 1) % 2;
```

This computation provides a cyclic pattern in that if *sideMove* is equal to one, the value of *sideMove* is updated to zero. If the value of *sideMove* is zero, it is computed as being one. This is how the chess game toggles between Player One's (WHITE=1) and Player Two's (BLACK=0) turns. This operation is effective in this style of tournament game because it is being hosted and played on one computer only.

Converting the chess game to NetChess required solving the problem of whose turn it was. One possible solution was to allow both computers to calculate the variable *sideMove*. The player on the server would take a turn and move a piece. When the move completed successfully, the *sideMove* on the server would be calculated. This would kick off a message to the client to update *sideMove*. Likewise, the server would not make a move on *sideMove* values that are zero. The client would send the server a message that it has completed its move. The server would update its *sideMove* value just as the client did before it sent its message. This toggling of *sideMove* values across the net is shown in Figure 10-24. There is a flaw in this logic that developers unfamiliar with client-server programming will not catch right away. With two nodes computing *sideMove* values based on messaging, who's to say that these values are synchronized?

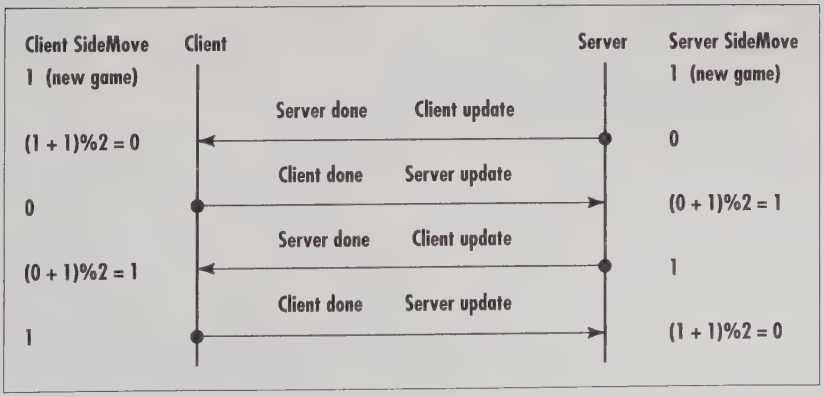


FIGURE 10-24
Independent player turn updating using messaging

A lot of novice developers fall into this simple pitfall. The solution is to let the server calculate and distribute the player moves. This scenario is a lot like the children's game Red Light, Green Light. On a red signal, the client's game processing activities are suspended and on a green signal, they are resumed. The server suspends its game activities, but not its server duties, on a red light issued to itself. The server architecture now resembles the operations in Figure 10-25.

In this model you don't worry about player turn synchronization. The server determines whose turn it is and notifies them. When it's the server's turn, the client is notified not to take a turn. Player One on the server makes a move. When the move is completed, the server notifies itself not to move by setting a Boolean flag value, *myTurn*, to FALSE. The client is then notified that it can take a turn. The value, *myTurn*, on the client, is set to TRUE, letting Player Two take a turn. The Boolean flag is a token value that allows use of the same chess game code on both the client and the server nodes. Whoever holds the token is allowed to make a move. The token is solely issued and updated by the server.

There are two specific cases to handle when the server determines whose move it is. One, it's the client's turn, and the client receives notification. Second, it's the server's turn to move. Essentially, you need to design the code to *host* the game. The code found in Listing 10-30 is used by NetChess to determines the player turns.

LISTING 10-30 The game host server

```
void NCMontyHall()
{
  // if Server turn complete call Monty
  if (bGameServer == TRUE)
  {
    sideMove = (sideMove + 1) % 2;
    NCTurnNotify();  // send the client sideMove value
    if (sideMove == SERVER_MOVE)
      myTurn = TRUE;
   else myTurn = FALSE;
  }
  else // client called Monty
  {
    if (sideMove == CLIENT_MOVE)
      myTurn = TRUE;
    else
      myTurn = FALSE;
  }
  processScreen = TRUE;
}
```

Our virtual game host, NetChess (NC) MontyHall, handles the distribution and updates of player turns. NCMontyHall handles the two instances of whether the node is a client or a server. Notice here that the notion of a WHITE

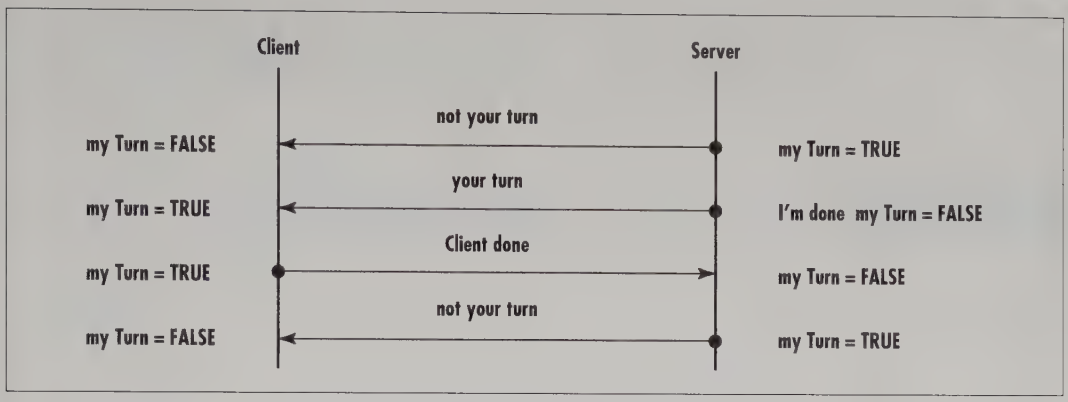


FIGURE 10-25 Server hosting a tournament game

board player and a BLACK board player have been abandoned. We now have a SERVER_MOVE and a CLIENT_MOVE.



Server Passes Turn to Client

Monty checks to see if the node is a client or a server by examining the *bGameServer* flag (remember this flag from the DPTOOLS source?). If *bGameServer* is set to TRUE, Monty is executing on a server. In this case, NCMontyHall is called because the server has just completed its move. Monty computes the new player turn and notifies the client of this change. The server calls the NetChess routine NCTurnNotify(). In the NetChess code, all functions that send a DirectPlay message have an *NCDoSomethingNotify()* naming convention. The code for NCTurnNotify() is shown in Listing 10-31.

LISTING 10-31 Notifier for whose move it is in the chess game

```
void NCTurnNotify()
{
  LPNETCHESSMSG lpMsg;
  int                              msgLen;

  lpMsg = (LPNETCHESSMSG) dpMsgBuf;
  lpMsg->msg = MSG_PLAYER_TURN;
  lpMsg->whoseMove = sideMove;

  msgLen = sizeof(NETCHESSMSG);
  lpDP->Send( dpID, dpNodeID, 0, (LPSTR) dpMsgBuf, msgLen);
}
```

This messaging looks a lot like the LPHELLOMSG data structure used in the previous example of messaging. Remember you can pretty much put

anything in a message as long as you limit the message size to 256 bytes. As a rule of thumb for network design, the smaller the message the better. You have a lesser chance of fragmentation over a busy network. The declaration of the *NetChess* message is shown in Listing 10-32.

LISTING 10-32 The *NetChess* NETCHESSMSG structure

```
#define MSG_PLAYER_TURN   0x40

typedef struct
{
  BYTE      msg;
  BYTE      whoseMove;
  BYTE      piece;
  BYTE      color;
  BYTE      posx;
  BYTE      posy;
  BYTE      onBoard;
  BYTE      updateScreen;
  DWORD dpid;
} NETCHESSMSG, *LPNETCHESSMSG;
```

The recipient of the message MSG_PLAYER_TURN is the client node. The client is identified by the DirectPlay ID *dpNodeID*. The player turn value is placed in the message at field *whoseMove*. The message is read by the client using a modified function GameMsg() from the DPTOOLS source. The completed source to the NetChess function GameMsg() can be found in Listing 10-33.

LISTING 10-33 Message processing used in *NetChess*

```
void GameMsg(int msg, DWORD msgLen)
{
  switch(msg)
  {
    case MSG_SERVER_HELLO:
      break;
    case MSG_CLIENT_HELLO:
      dpNodeID = ReadHELLOMsg();
      SendACK(MSG_SERVER_ACK, dpNodeID);
      break;
    case MSG_SERVER_ACK:
      break;
    case MSG_CLIENT_ACK:
      break;

    // handle NetChess specific messages
    case MSG_PLAYER_TURN:
      NCReadTurn();
      break;
    case MSG_GRAB_PIECE:
```

```
      NCReadGrab();
      break;
    case MSG_RELEASE_PIECE:
      NCReadRelease();
      break;
    case MSG_MOVE_COMPLETE:
      NCReadMoveComplete();
      break;
    case MSG_CAPTURE_PIECE:
      NCReadCapture();
      break;
    default:
      break;
  } // switch msg
}
```

Monty sends the message MSG_PLAYER_TURN to the client. The client acts on the message using the modified DPTOOLS function GameMsg(). When GameMsg() sees the message MSG_PLAYER_TURN, it calls the function NCReadTurn(). All NetChess functions that read a message use the calling convention NCRead*Something*(). The code for NCReadTurn() is shown in Listing 10-34.

LISTING 10-34 Reading player turn message

```
void NCReadTurn()
{
  LPNETCHESSMSG lpMsg;
  BYTE          msg;

  lpMsg = (LPNETCHESSMSG) dpMsgBuf;
  msg = lpMsg->msg;
  sideMove = lpMsg->whoseMove;

  // retrieve client turn
  if (bGameServer == FALSE)
  {
    if (sideMove == CLIENT_MOVE)
      myTurn = TRUE;
    else
      myTurn = FALSE;
  }
}
```

This function is simple in design. Whenever the client reads a MSG_PLAYER_TURN message, it examines the *whoseMove* field and looks to see if the value is CLIENT_MOVE. If it is, the *myTurn* flag is set to TRUE. If there is any other value in the *whoseMove* field, the *myTurn* flag is set to FALSE. The client doesn't make a move unless the server specifically tells the client to make a move.



The Client Notifies the Server of a Completed Turn

The messaging to this point is one sided. There is a mechanism for the server to notify the client it is done and hand over the player move token. This allows for at most two complete plays in a game before it freezes up. Why is this? The client and the server don't know what to do after the client completes its player move. In the NetChess code, when the player completes the move on his node, the function releasePiece() determines whether the player made a legal move or not and updates the player screen. Add a few lines of code at the end of this function for the client and server nodes to start their messaging of play move completion. This looks like,

```
if (bGameServer == TRUE)
  NCMontyHall();
else  // client move complete call Monty on server
  NCMoveCompleteNotify();
```

This code examines the *bGameServer* flag and determines whether you're executing on the client or server node. If *bGameServer* is TRUE, you're on the server node and you execute NCMontyHall() at the completion of your move. In the event *bGameServer* is set to FALSE, you are executing on the client node. The client doesn't call NCMontyHall(). It sends a message to the server signaling the player has successfully completed a move. The code for the client notifier NCMoveCompleteNotify() is found in Listing 10-35.

LISTING 10-35 Notifying the server of player move completion

```
#define MSG_MOVE_COMPLETE 0x43

void NCMoveCompleteNotify()
{
  LPNETCHESSMSG lpMsg;
  int msgLen;

  if ( (bGameServer == FALSE) && (myTurn == TRUE))
  {
    lpMsg = (LPNETCHESSMSG) dpMsgBuf;
    lpMsg->msg = MSG_MOVE_COMPLETE;
    msgLen = sizeof(NETCHESSMSG);
    lpDP->Send( dpID, dpNodeID, 0, (LPSTR) dpMsgBuf, msgLen);
  }
}
```

This code only sends the message MSG_MOVE_COMPLETE and no other values. Notice in this case, the DirectPlay ID *dpNodeID* is the server DPID which was picked up using the simple LPHELLOMSG and ACK from the earlier messaging example. The server reads this message using the same GameMsg() function. When the server reads the message MSG_MOVE_COMPLETE, GameMsg() calls the function NCReadMove(). This looks like Listing 10-36.

LISTING 10-36 Reading the player move completion message

```
void NCReadMoveComplete()
{
  LPNETCHESSMSG lpMsg;
  BYTE          msg;

  lpMsg = (LPNETCHESSMSG) dpMsgBuf;
  msg = lpMsg->msg;

  // call game host server to update move
  NCMontyHall();
}
```

, In this code, the server has been notified that the client has completed its move. The server calls NCMontyHall() to update the value of the server's *sideMove* variable and distribute the value back to the client. The server updates the *myTurn* flag to TRUE in the NCMontyCode() and the token is passed to the server to commence a move. The client sits on the other end of the wire waiting for the whoseMove token to notify that it's the client's turn again.

The code for determining the player's turn is the most complicated of the messaging in NetChess. If you understand the token passing that's going on between the two nodes, then you can understand the other messaging. Just perform greps on NC tags in the CHESS.CPP code. There aren't a lot. Then piece together the code in the DPLAYX.H file.



Networking Is the Key to Global Domination

With the Internet open to the public, the world has certainly become a smaller place. UNIX workstations have a game called *X-Pilot* were players anywhere in the world can join in to attack other players with their spaceships. The concept of modem dial-up playing on UNIX devices is considerably different from that of the game PC player, but I see that changing very shortly. Most developers consider modem play a connection between two players, most likely across town.

In the corporate world, the modem is a remote network link for sales people to dial-up and connect for remote client server activities, querying a sales database or retrieving their e-mail. These people use dial-up routers, like a Telebit Netblazer or have ISDN connections, on a daily basis. In the evening, during their off hours, these same people unwind with friends from work, using Point-to-Point Protocols (PPP) to provide IPX connections and Serial Line Internet Protocol (SLIP) that become service providers for games such as *DOOM* and *Hexen*. To these people, the modem is an extension of the corporate network to their homes. To game developers, this means providing products to customers who may already be networked.

The world is changing overnight. The nineties will be known as the era of the telecommunication's revolution. Game developers are going to be an integral part in this. Cable channels now offer SEGA game connections to subscribers. It won't be long before some entrepeneur figures out how to offer a virtual reality world service to anyone with a modem or an ISDN connection and a valid credit card.

11

THE DUNGEON

11 THE DUNGEON

Chapter 11 is a marriage of all the previous chapters. You'll see something old, something new, something borrowed, and a few pixels that are blue. In this chapter, you'll design a role-playing game (RPG) called *Dungeon* that will utilize the Direct X technology of DirectDraw, DirectSound, DirectInput, and DirectPlay. A lot of these concepts will look familiar to you. You're essentially dumping everything you have used so far in *Spells of Fury* into a mixing bowl and creating a game.

EPISODE ESCAPE



"Bary Maelstrom, you have stood in the way of the gods far too long. I would destroy you but your service to me as my slave would be invaluable," the plasma image of Agamemnon decreed.

"I will never serve you!"

"Bow down to me," Agamemnon ordered. Bolts of electricity shot from the eyes of the floating head. The intense energy struck me, knocking me to the ground, throwing me into a near-seizure. The muscles in my virtual body locked up in intense pain.

"In this room, I am supreme. You cannot change into any virtual form here. My slaves are coming for you. Soon you will be one of my legion," the sorcerer decreed.

I was trying to fight the intense pain that raced through every nerve in my body. I couldn't just lie on the floor and let Agamemnon win. I turned my head toward the hall I came from. A rush of air was blowing in. It felt like the rush of air in a subway station as an oncoming train pushed the air out of the tunnel. The volume of air steadily increased as whatever was pushing it came closer. I noticed that Agamemnon's attention was diverted to the hall as well.

Just then, I realized it wasn't Agamemnon's legion of virtual zombies that was coming. It was Zero. My spirits elevated from despair to a new hope. A shrieking echoed out of the tunnels as the Zero Virus vaporized the legion in her path.

"Ya hear that? You're not going anywhere, pal!" I yelled at the 3D display of Agamemnon's face. The sorcerer snarled at me.

Plasma bolts shot into the chamber. At this moment I was thankful I was lying on the ground. They would have burned right through me if I were still standing.

Two bolts of energy struck Agamemnon head-on. The energy wrapped around the virtual face of the sorcerer and burned gaping holes in his pixel integrity. Agamemnon appeared to be in pain. Four more plasma bolts were hurled into the chamber and struck Agamemnon.

I lay there completely helpless as the encroaching winds grew in strength. Zero was near. I looked up and saw Agamemnon's face fade from the console. I had to pull myself to the console to open a port to get out. Fearing for my own hide, my plan of attack was to vacate the premises. A deep chuckle rumbled through the chamber. I didn't know if it was my natural sense of survival or my years of reading comic books, but I was sure it was the treacherous last laugh of a supervillain. In seconds, my greatest fears were realized.

A form appeared before me, which was...me.

"What do you think? I am the new Bary Maelstrom. And you? You will die in here with your friend Carol. It was awfully nice of you to lend me your true body. In it I will take over the world. And the new gods will rule the earth." Thus Agamemnon revealed his diabolical plan. The sorcerer mockingly waved to me his last farewell, just as he was about to step through the port that I needed to go home.

"It'll never work! The Grinch will shoot you down in cold blood. He never intended for me to leave," I said, pointing out the obvious flaw in his plan.

"That's correct. You were never intended to leave. I summoned you here. The one you call the Grinch is under my control. Now if you'll excuse me, I'm really anxious to return to the real world."

Agamemnon's steps were hindered by powerful electric arcs from Zero, pulling him back away from the portal.

"Bary, quickly, step through the port. Only one person may leave, and then the port is shut forever." Zero spoke in Carol's soft voice.

"I can't move. Agamemnon paralyzed me."

Zero walked over to me. I could see the eyes of Carol gazing upon me. She reached down and touched my head. I felt a surge of information fill my mind. In a millisecond, I saw the entire life of Carol Goldmen, as well as information on Nazi rocket engines, alien propulsion systems, and other secrets of time hidden in the artifact. The surge revitalized my virtual body. I could feel my toes again.

Carol lifted me to my feet and gave me a farewell hug.

"Bary, you must hurry. I kicked off the self-destruct mechanisms in here. Agamemnon shall be no more. But most important, you have the information in you to stop the contingency factor in 13." Carol's soft voice faded as the Zero Virus resumed its form. Zero blasted me through the port and I was hurled back to my reality.



The Dungeon Map

In Chapter 9, you used a terrain map to extend the width of your 640-pixel-wide game screen out to *n* pixels. The terrain was a two-dimensional map that was 10 rows by *n* cells. Each entry in the terrain map was an integer identifying which tile to draw if this slice in the map was visible on the display screen. The assumption for the terrain map is that the height of the screen is constant, 480 pixels or 15 cells, and the width is greater than or equal to 640 pixels or 20 cells. Using the relationship that each tile is 32 x 32 pixels, you have 32 pixels equaling 1 cell in the terrain map. The terrain map provided you with the mechanism to scroll the Bubonic screen horizontally. Now you're going to design a map that will allow you to scroll horizontally and vertically in the *Dungeon* game.



A Horizontal and Vertical Scrolling Terrain

The vertical movement for *Bubonic the WartHog*, in Chapter 9, was bounded by the fixed screen height of 640 pixels. In *Dungeon*, players can maneuver their warriors through an uncharted dungeon to the north, south, east, and west. The dungeon has a terrain map that is 60 x 60 cells. The logical screen is 15 rows by 20 cells. This means your new terrain map is three times as wide as your screen and four times as tall.

You use the same logical mappings to divide your 640 x 480 pixel DirectDrawSurface into a 15 x 20 screen, as shown in Figure 11-1. The logical screen has 15 rows (480/32=15), each with 20 cells (640/32=20). Each cell in the logical screen represents a 32 x 32 pixel region where you will draw a tile.

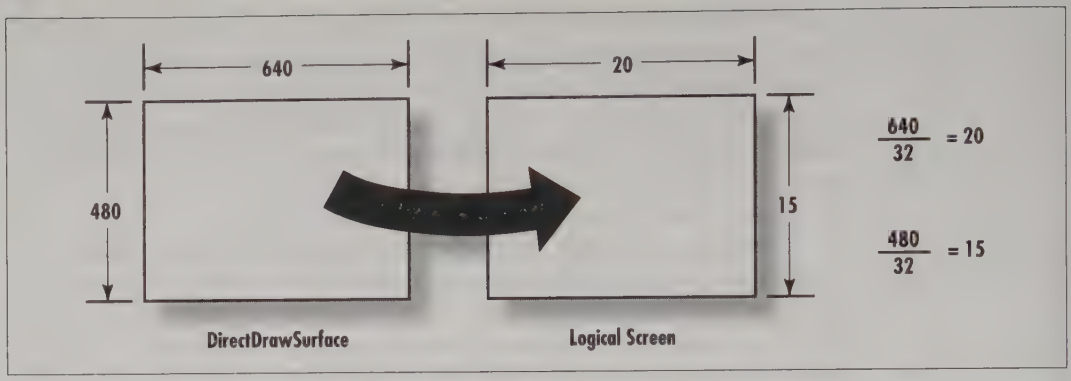


FIGURE 11-1 Logical screen for *Dungeon*

The Dungeon terrain map, shown in Figure 11-2, is 60 rows x 60 cells. This map is 12 times the size of your logical screen. If you were to allocate an offscreen DirectDrawSurface to accommodate this map, you would need one the size of 1920 x 1920 pixels. You have seen from previous examples (see Chapter 9) that an offscreen DirectDrawSurface isn't necessary for your terrain map. The decomposition of a DirectDrawSurface into map cells 32 pixels wide will suffice.



The Terrain Map Contents

In Chapter 9, in the *Bubonic the WartHog* example, your terrain map gave information about the tile as well as the tile type. The *Dungeon* terrain map will also contain information about the tile, but since this is a three-quarter-view tile game, you will need to know what objects and players are currently on this cell. Look at Figure 11-3. You used this style terrain map back in Chapter 3 for the PowerSuit example. Remember, the terrain map was an array of the *SPOT* structure. In *Dungeon* you'll cut and paste the *SPOT* structure from the PowerSuit source code into the DUNGEON.H header file. This declaration can be seen in Listing 11-1.

LISTING 11-1 Dungeon terrain map

```
#define TERRAIN_WIDTH 60
#define TERRAIN_HEIGHT_60
typedef struct {
  int cell;  // tile ID
  PLAYER *playrPtr;  // player standing on this tile
  ITEM *itemPtr;  // item on this spot
} SPOT;
SPOT terrain[TERRAIN_WIDTH][TERRAIN_HEIGHT];
```

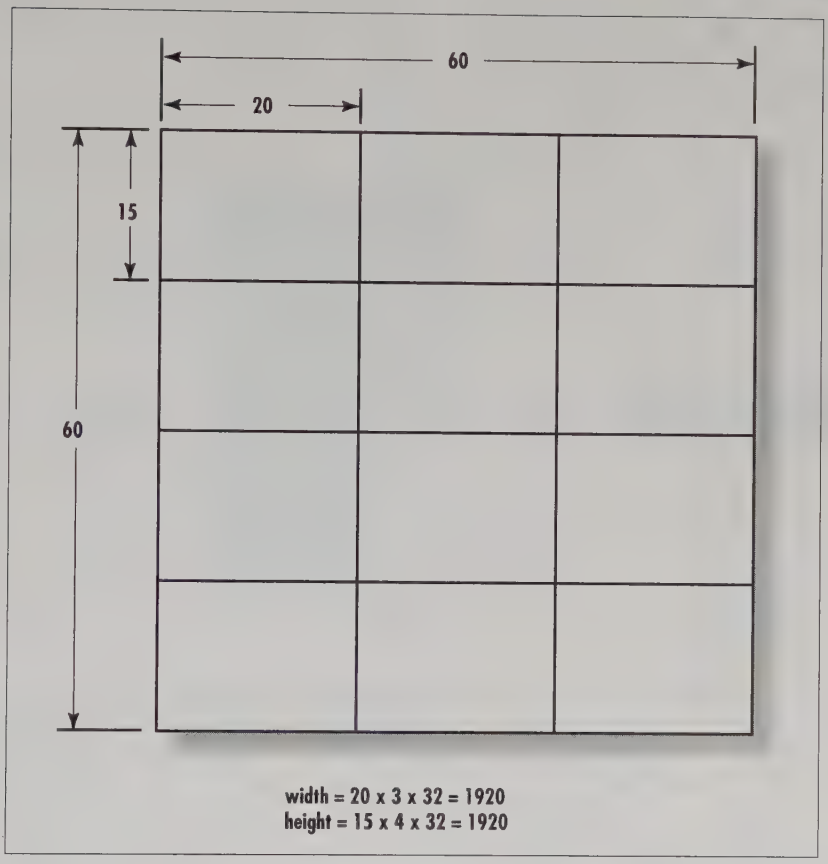


FIGURE 11-2

Dungeon terrain map

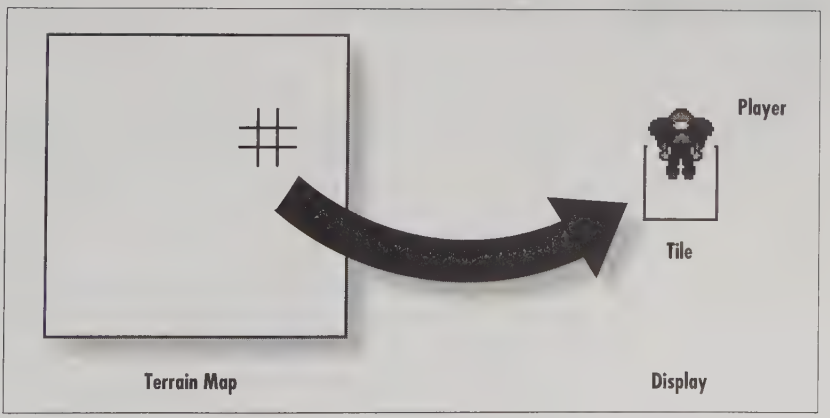


FIGURE 11-3

Terrain map cells

The terrain map is a 60 x 60 cell array of the *SPOT* structure. Each cell in the terrain array represents a tile. If a player or an item is on this tile, then the pointers reference the corresponding values; otherwise, the pointers are NULL.

Initializing the Terrain Map

The terrain map is stored in the file DUNGEON.INF. The INF file, like the previous INF files in *Spells of Fury*, contains integers for tiles as they are to be placed in the terrain map. Each integer is a tile type, as in the *Bubonic the WartHog* example, and each tile type in the terrain map is an index into a tile database. The tile database in *Dungeon* is an array of the *CELL* structure. The *CELL* structure definition is found in header file DUNGEON.H, as is shown in Listing 11-2.

LISTING 11-2 CELL structure declaration

```
typedef struct {
  int posx;          // top-left corner of tile
  int posy;          //  in offscreen DirectDrawSurface
  int height;      // height of tile
  int width;        // width of tile
  int frames;      // number of animation frames in sequence
  int visibility;    // line of sight
  int obstacle       // difficulty of passing obstacle 0-3
} CELL;
```

As with previous programming examples, tiles have attributes. Here, your tile attributes include the top-left corner of the tile as it is stored in the offscreen DirectDrawSurface, tile height and width, and tile characteristics such as visibility and obstacle difficulty. The tile *CELL* database in file DUNGEON.CPP is provided in Listing 11-3.

LISTING 11-3 Tile CELL database

```
CELL wall_cells[] = {
  33, 1, 76, 30, 1, 0, 0,  // tile type = 0  ground tile
  1, 1, 76, 30, 1, 3, 3,   // tile type = 1 wall tile
  1, 1, 76, 30, 1, 3, 0,   // tile type = 2 secret door
  65, 1, 76, 30, 1, 3, 3,  // tile type = 3 column
  1, 79, 76, 30, 1, 0, 3,  // tile type = 4 wall with bars
};
```

You'll notice that tile types 1 and 2 reference the same tile in the offscreen DirectDrawSurface but they have different tile attributes. Tile Type 1 is a wall tile that is impassable, and its obstacle value is 3. Tile Type 2 is a secret door tile; its obstacle value is 0. For the sake of simplicity, you'll only use five tiles in this example. These five tiles will populate the 60 x 60 cell terrain map. As an exercise to understand where the *posx* and *posy* values came from, open the DUNGEON.BMP file with the Microsoft Windows 95 Paint program found in the Accessories directory. Move the cursor in the Paint program to locate the upper left corner of the .BMP file. You should see the tiles in Figure 11-4. These are your five tiles you'll use to draw your terrain.

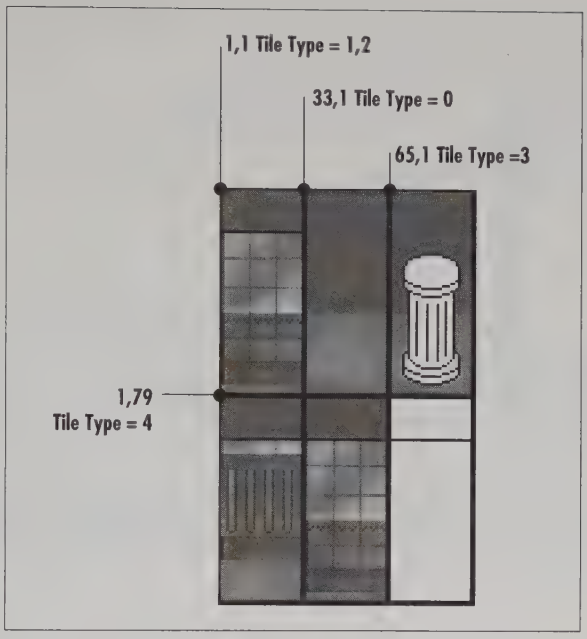


FIGURE 11-4
Terrain tiles and their cell values

To read in a terrain map from a file, use the function initMap(), in file DGNGAME.CPP. The function is very similar to the map initializer you used in Chapter 9. The DUNGEON.INF is a text file with space-delimited values. The values are space-delimited so you can use the C run-time library function fscanf() to read in your terrain map. Unlike the *Bubonic the WartHog* INF reader, this file contains no control characters such as, "\t" or "\n". It's a flat file. This makes it a little hard to read when you open it with the DOS edit utility. But nevertheless, go ahead and take a look at the file. The code to read the INF file is presented in Listing 11-4.

LISTING 11-4 Reading the DUNGEON.INF file

```
BOOL initMap()
{
  FILE *stream;
  int i,j;

  // Open the map.inf file for reading.
  if(!(stream = fopen( "MAP.INF", "r")))
  {
    return FALSE;
  }

  // Go through the file and initialize the terrain array.
  for (i = 0; i < TERRAIN_HEIGHT; i++)
```

continued on next page

continued from previous page

```
    for(j = 0; j < TERRAIN_WIDTH; j++)
    {
      fscanf(stream,"%d",&terrain[i][j].cell);
      terrain[i][j].playerPtr = NULL;
      terrain[i][j].seen = FALSE;
      terrain[i][j].itemPtr = NULL;
    }
    fclose(stream);
    return TRUE;
}
```

The terrain map is an array of the *SPOT* structure. Even though you are reading a flat file, you loop through the 60 cells on all 60 rows of the terrain map to initialize it. Each terrain cell is initialized to the corresponding tile type from the INF file. Remember, the terrain cell also maintains information about what item or player is standing on the current cell. You haven't initialized the player or the item data yet, so you initialize these fields with a NULL pointer. I'll discuss the *PLAYER* and *ITEM* structures in a moment. For now, just keep in mind that these are linked lists, one for the *PLAYER*(s) and the other for the *ITEM*(s) on the terrain map.



Locating a Tile in the Map

The key to using the terrain map is understanding how to translate which tile is to be drawn where. The terrain map is a logical representation of how you want your tiles laid out when they're drawn to the display. The representation is logical because you don't use 1920 rows x 1920 cells to represent each pixel in the map. Instead, an integer represents a 32 x 32 pixel region, called a tile. This correlation, as you have seen, reduces your terrain map to only 60 x 60 cells. Therefore, when you retrieve an integer value from the terrain map, you are retrieving a value indicating the type of value located in that cell, as shown in Figure 11-5.

The tile type is the index into the *wall_cells* tile lookup table, (see Figure 11-6). It is the information in the tile lookup table that provides you with the information that the *Spells of Fury* DDTOOLS function drawOneToBack() will need for blitting. Recall that your function drawOneToBack() requires the destination *x* and *y* coordinates for drawing to *lpDDSBack*, the DirectDrawSurface used for flipping. Also, a source RECT is required of the source sprite. The source sprite is your tile on the offscreen DirectDrawSurface, *lpDDSOne*. The coordinates for the source RECT structure can be set equal to the first four fields in a row of the *wall_cells* lookup table. A tile source RECT can be retrieved using the code in Listing 11-5.

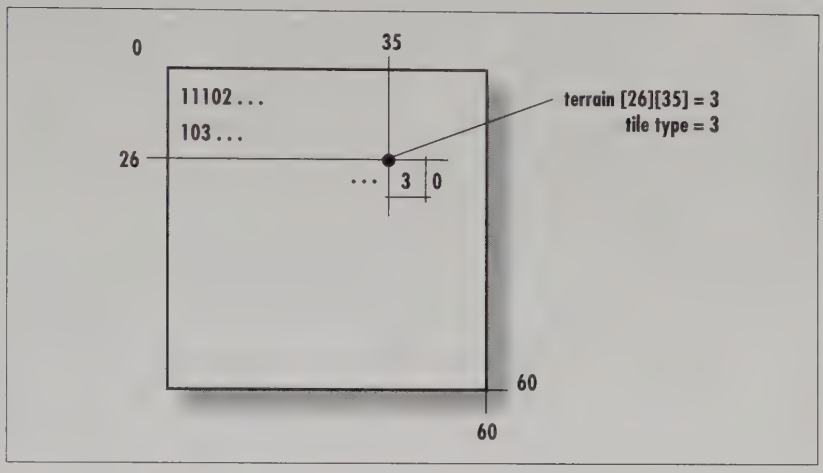


FIGURE 11-5
Retrieving a tile type from the terrain map

LISTING 11-5 Locating the tile on the offscreen surface

```
RECT srcRect;
int row = 26, cellIndex = 35;
int tileIndex;

tileIndex = terrain[row][cellIndex].cell;
srcRect.left = wall_cells[tileIndex].posx;
srcRect.top = wall_cells[tileIndex].posy;
srcRect.right = srcRect.left + wall_cells[tileIndex].width;
srcRect.bottom = srcRect.top + wall_cells[tileIndex].height;
```

The tile type in the terrain map is the index into the tile lookup table. When you break it down as you did in the code above, you retrieve the tile type from the *wall_cells* lookup table and you compute the tile source RECT structure to

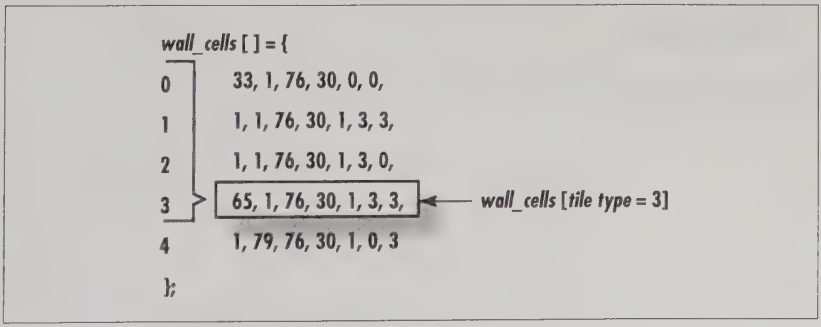


FIGURE 11-6
Using the *wall_cells* lookup table

use in the blit operation. The code in the *Dungeon* game doesn't appear this straightforward, as you can see below:

```
srcRect.left = wall_cells[terrain[row][cellIndex].cell].posx;
srcRect.top = wall_cells[terrain[row][cellIndex].cell].posy;
srcRect.right = srcRect.left +
➥wall_cells[terrain[row][cellIndex].cell].width;
srcRect.bottom = srcRect.top +
➥wall_cells[terrain[row][cellIndex].cell].height;
```

If you take a close look at the two code snippets, their function is identical. Rather than using a variable *tileIndex* the terrain cell is de-referenced on the fly. This is just a coding convention that's used in the *Dungeon* program. This is all the information from the tile lookup tables you need to call drawOneToBack(). Just keep in mind when reading code, that when you see *wall_cells* and a Windows RECT structure in the same line of code, you are retrieving a tile for a blit operation.

Now that you understand the architecture of the terrain map and the tile lookup table, you're ready to add some players and items to it. How you do this will be the topic of the next discussion.



Designing for Multiple Players

Dungeon is a two-player game with a DirectPlay interface for network play. The numeric keypad with the numlock on or a game controller may be used to navigate your player through the dungeon. To keep track of how many players you have and what player is located where, you use a linked list. Now obviously, since you only have two players, a linked list is a bit of an overkill. Just keep in mind that this is designed for network play, and you can modify the game for up to 16 players.



The Dungeon Player

When *Dungeon* players maneuver their warriors through the dungeon, they will encounter objects, walls, secret doors, and hostile opponents. The objects a player may pick up include swords, shields, and potions. Objects such as these, when used, may affect the player's warrior or an opposing player. The swords and potions indicate that each warrior has a state of health. Potions can be used on opponents to decrement their health, or a potion can backfire. Objects that increment the player's health may also be picked up. All of these actions affect attributes of the player's warrior. While all this is happening to the player's warrior, he or she could be inflicting damage on another opponent. This is why you will use a linked list to maintain what is happening to what player and where.



The PLAYER Linked List

Dungeon is a distributed computing environment game. The game is being played over a network. Each *Dungeon* game player has an instance of a linked list that maintains what is happening with each game player. The list elements are updated in real time. You won't be using a game server over the net, such as the function NCMontyHall from *NetChess*. The *PLAYER* linked list will maintain information about the game player on the current instance of the game as well as the remote instance. When a player attacks an opponent, the current instance of the linked list is updated and messages are sent to the remote player to update his or her instance. The *Dungeon* game uses the *PLAYER* structure as a node in its linked list. Remember you could add up to 16 players to this game using a list. I am going to keep this discussion localized to a current game player instance and one remote game player. This simplifies the explanations. So when I mention a remote player on the net, think of it as remote player *n*. The *PLAYER* structure for *Dungeon* is shown in Listing 11-6.

LISTING 11-6 The *PLAYER* structure used in *Dungeon*

```
typedef struct player {
  int Posx;           // current x position in the terrain map
  int Posy;           // current y position in the terrain map
  int facing;         // line of sight
  int cellPosx;       // left corner of current player sprite
  int cellPosy;       // top corner of player sprite
  BOOL alive;         // is our player alive
  int armor;          // current armor points the player has
  int maxArmor;       // maximum armor points a player may have
  int health;         // current health points of the player
  int maxHealth;      // maximum number of health points
  int attack;         // how many attack points
  int maxAttack;      // maximum number of attack points
  ITEMS *itemsCarried[8]; //player can carry up to eight items
  BOOL firing;        // player is attacking
  BOOL weaponFired;   // weapon used
  float timeModifier; // time modifier
  DWORD stopTimeModifier; // stop of player time modifier
  player *nextPlayer; // pointer to next player in the list
}PLAYER;
```

The *PLAYER* list maintains quite a bit of information about each player. The *Dungeon* game's designer, Mike Thiels, is an RPG enthusiast, so you'll see a few fields, such as health, armor, and attack, that reflect his interests to port these games over to a computer. Included in the *PLAYER* list node are an array of items the player is in possession of and time delay modifiers for network messaging. The facing field is an interesting attribute Mike included. Since the game is network play, you can't see your opponent unless he or she is in your

line of sight. If your opponent is not facing you, you can sneak up and attack them.



Initializing the PLAYER List

In the *Dungeon* game example, a *PLAYER* list is composed of two game players, the player on the local instance of the game and the player on the remote instance of the *Dungeon* game. Both instances of the game must initialize a local *PLAYER* list. The game *Dungeon* uses the function initPlayer() to build the local instance *PLAYER* list. The code for initPlayer() is presented in Listing 11-7.

LISTING 11-7 Initializing the *Dungeon PLAYER* linked list

```
BOOL initPlayer(int playerNumber)
{
  PLAYER     *tempPtr, *endPtr;

  while (playerNumber != 0)
  {
    // If no players in list then make head pointer point to
    // first players structure
    if (playerHead == NULL)
    {
      // Alloc the memory for PLAYER structure.
      if (!(playerHead = (PLAYER*)LocalAlloc( LPTR, sizeof(PLAYER))))
        return FALSE;
      // Head of list, next is NULL.
      playerHead->nextPlayer = NULL;
      // Fill in the player's structure.
      fillPlayerInfo(playerHead,playerNumber);
    }
    else
    {
      // Make endPtr point to last player structure in
      // the link list
      endPtr = playerHead;
      while (endPtr->nextPlayer != NULL)
        endPtr = endPtr->nextPlayer;

      // Allocate memory for new player struture.
      if (!(tempPtr = (PLAYER *)LocalAlloc( LPTR, sizeof(PLAYER))))
        return FALSE;
      // Link the new structure to the list.
      endPtr->nextPlayer = tempPtr;
      // Make new structure end of list.
      endPtr = tempPtr;
      // Fill in the new structures information.
      fillPlayerInfo(endPtr, playerNumber);
```

```
    }
    playerNumber -= 1;
  }
  return TRUE;
}
```

The function initPlayer() builds your *PLAYER* linked list. From Chapter 7, you know you have two conditions for building a list: one, the list is empty; or two, at least one node is in the list. The body of the initPlayer() code is primarily a while loop. With each pass of the loop you decrement the variable *playerNumber*. The loop terminates when *playerNumber* reaches 0. The while loop handles the two cases for your list initialization. When initPlayer() is called from your WinMain() function, the *PLAYER* list is empty. The global *PLAYER* list pointer, *playerHead*, references the start or head of your list. The *playerHead* list pointer is initialized to NULL, meaning your list is empty (see Figure 11-7(a)). When the while loop executes in initPlayer(), it will first examine the pointer *playerHead* to determine which of the two conditions it must handle to insert a *PLAYER* node into the list. The code you'll use to insert into an empty list is shown in Listing 11-8.

LISTING 11-8 Inserting a *PLAYER* into an empty list

```
if (playerHead == NULL)
{
  if (!(playerHead = (PLAYER *)LocalAlloc( LPTR, sizeof(PLAYER))))
    return FALSE;
  playerHead->nextPlayer = NULL;
  fillPlayerInfo(playerHead,playerNumber);
}
```

If the *PLAYER* list is NULL, you are inserting the first *PLAYER* node into the list. You allocate a new *PLAYER* node using the Windows API call LocalAlloc, as illustrated in Figure 11-7(b). If the memory was allocated successfully, pointer *playerHead* is assigned to the new *PLAYER* node. This node is now the first *PLAYER* node in the list. Since you only have one node in the list, you assign the *playerHead->nextPlayer* pointer a NULL pointer, as illustrated in Figure 11-7(c). You can now instantiate the node with the game player's information using the function fillPlayerInfo(), shown in Figure 11-7(d). That's all you have to do to initialize the *PLAYER* list.

The second condition your while loop must handle is how to insert into a list that isn't empty. If the *playerHead* pointer isn't NULL, then you have at least one node in the list, as is shown in Figure 11-8(a). Keep in mind the *Dungeon* example program only has two players, but the list represents the conditions for two or more players. The code used by the while loop for this condition is provided in Listing 11-9.

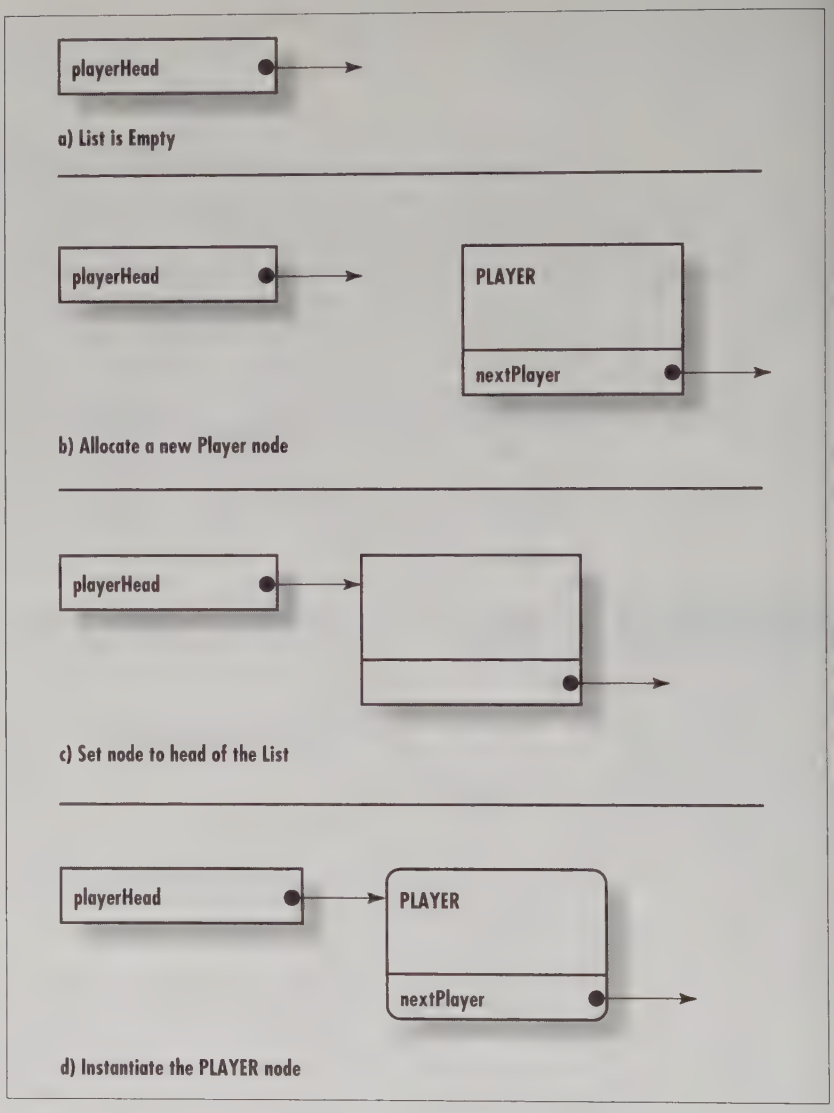


FIGURE 11-7
Inserting a *PLAYER* node into an empty list

LISTING 11-9 Inserting a PLAYER into a nonempty list

```
else // playerHead != NULL
{
  endPtr = playerHead;
  while (endPtr->nextPlayer != NULL)
    endPtr = endPtr->nextPlayer;
  if (!(tempPtr = (PLAYER *)LocalAlloc( LPTR, sizeof(PLAYER))))
    return FALSE;
  endPtr->nextPlayer = tempPtr;
  endPtr = tempPtr;
  fillPlayerInfo(endPtr, playerNumber);
}
```

Since the list is populated by one or more nodes, you will need an additional *PLAYER* node pointer, *endPtr*, to walk the list and locate the last node. Using a while loop, you walk the list node to node until the pointer *nextPlayer* is NULL, as is shown in Figure 11-8(b). Once the end of the list is located, you can allocate a new *PLAYER* node to insert at the end of the list. The new *PLAYER* node is referenced by a temporary node pointer, *tempPtr*, shown in Figure 11-8(c). To insert the new node at the end of the list, assign the pointer *endPtr->nextPlayer* to the new node. Now the last node in the list references the new *tempPtr* node. You now want to update the new node to be the last node in the list, so you assign the pointer *endPtr* the address of *tempPtr*, as shown in Figure 11-8(d). The new node is now inserted in the list. You can now instantiate the node with player information, as shown in Figure 11-8(e).



Initializing the PLAYER Node Data

When I discussed inserting a player into a list, you always had for your last step instantiating the *PLAYER* node. Instantiation means you are going to assign real data to an already initialized data structure. The function you will use to do this is fillPlayerInfo(). This function will randomly place the warrior on the terrain map, and initialize armor, items, and the location of the sprite cells on the offscreen DirectDrawSurface, *lpDDSOne*. The function fillPlayerInfo() is shown in Listing 11-10.

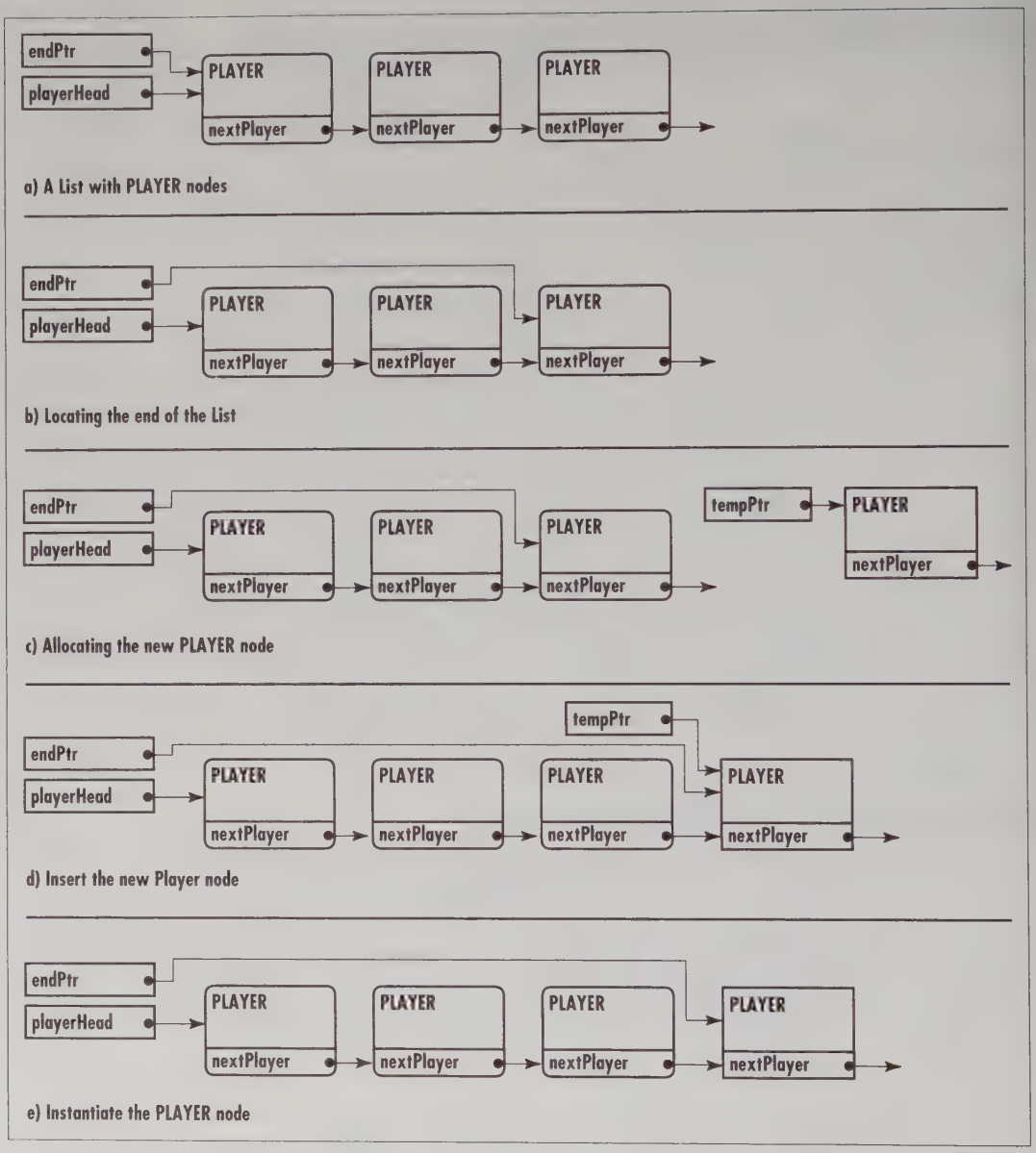


FIGURE 11-8 Inserting a *PLAYER* into an existing list

LISTING 11-10 Initializing the PLAYER fields

```
void fillPlayerInfo(PLAYER *playerPtr, int playerNumber)
{
  BOOL    posFound;  // Used for putting player in map.
  int     i;

  // player is not currently firing.
  playerPtr->firing = FALSE;
  // Search the map array for a suitable cell to place
  // the player
  posFound = FALSE;
  while (!posFound)
  {
    int iPosx = playerPtr->Posx;
    int iPosy = playerPtr->Posy;
    // Randomly determine where to put warrior
    playerPtr->Posx = rand() % TERRAIN_WIDTH;
    playerPtr->Posy = rand() % TERRAIN_HEIGHT;
    // If cell does not have another dungeon warrior, or a
    // solid obstacle then it's OK for placement.
    if ( (wall_cells[terrain[iPosy][iPosx].cell].obstacle < 03)
        && (terrain[iPosy][iPosx].playerPtr == NULL))
      posFound = TRUE;
  }
  // Have the cells playerPtr point to the powersuit.
  terrain[iPosy][iPosx].playerPtr = playerPtr;

  // The left point where the warriors graphic cells are.
  if (playerNumber == 2)
  {
    playerPtr->cellPosx = 124;
    playerPtr->cellPosy = 0;
  }
  else
  {
    playerPtr->cellPosx = 124;
    playerPtr->cellPosy = 376;
  }

  // Player has 2 armor points.
  playerPtr->armor = 2;
  playerPtr->maxArmor = 2;
  playerPtr->health = 2;
  playerPtr->maxHealth = 2;
  playerPtr->attack = 2;
  playerPtr->maxAttack = 2;

  // Player holds no items at start.
  for (i=0;i<8;i++)
    playerPtr->itemsCarried[i] = NULL;
  // Setup the flags for the powersuit.
  playerPtr->alive = TRUE;
  playerPtr->weaponFired = FALSE;
  playerPtr->timeModifier = (float)1.00;
  playerPtr->stopTimeModifier = 0;
}
```

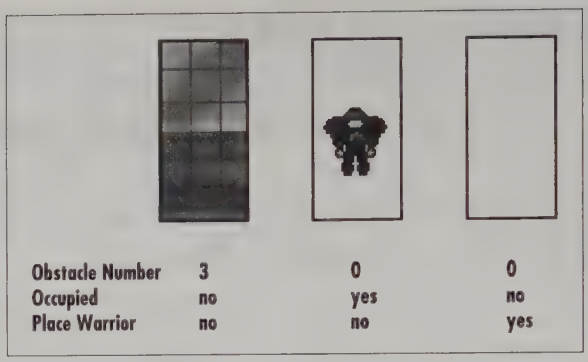


FIGURE 11-9
Placing a warrior in a terrain map cell

The function fillPlayerInfo() is where you initialize the warrior placement on the terrain map. The position is randomly generated so you must check to make sure that no warrior is currently occupying that position. Also, you must check to ensure that you don't initialize a warrior into a terrain map cell containing a wall or an obstacle. In Figure 11-9, you see the conditions for placing a warrior in a terrain map cell. The first cell in Figure 11-9 is a wall; its obstacle value is 3. You can't place your warrior there. In the cell next to it, you see the obstacle value is 0, but the cell is currently occupied by an opponent. So you can't place your player in this cell either. The last cell in Figure 11-9 has an obstacle value of 0 and the cell is unoccupied. You place your warrior here. The code to determine cell placement will loop until you find a cell that the warrior can occupy. The loop is terminated when the flag *posFound* is set to TRUE. The code for determining the *PLAYER* terrain map cell placement is shown in Listing 11-11.

LISTING 11-11 Placing a *PLAYER* on the terrain map

```
while (!posFound)
{

  playerPtr->Posx = rand() % TERRAIN_WIDTH;
  playerPtr->Posy = rand() % TERRAIN_HEIGHT;
  int iPosx = playerPtr->Posx;
  int iPosy = playerPtr->Posy;

  if ( (wall_cells[terrain[iPosy][iPosx].cell].obstacle < 03)
    && (terrain[iPosy][iPosx].playerPtr == NULL))
    posFound = TRUE;
}
terrain[iPosy][iPosx].playerPtr = playerPtr;
```

When the loop terminates, you have found a cell. The terrain map database must be updated with the warrior's present position. Recall that each cell in the map has a pointer to a *PLAYER*. If a *PLAYER* occupies this cell, the cell's *PLAYER* pointer is assigned the address of the *PLAYER* occupying the cell. Keep in mind



FIGURE 11-10
Sprite cells for playerNumber = 2

that the terrain map is kept in synch with the warrior's movement. When the warrior leaves this cell to move to another, the previous cell assigns the *PLAYER* pointer to NULL to indicate the cell is unoccupied. The new cell that the warrior occupies then assigns its *PLAYER* pointer to the occupying warrior's *PLAYER* address.

When you initialize the *PLAYER*, you must also initialize the *cellPosx* and *cellPosy* fields. These are the locations of the top-left corner of the warrior sprite, on the offscreen DirectDrawSurface, as shown in Figure 11-10. The code for this is shown in Listing 11-12.

LISTING 11-12 Initializing the PLAYER cell positions

```
if (playerNumber == 2)
{
  playerPtr->cellPosx = 124;
  playerPtr->cellPosy = 0;
}
else
{
  playerPtr->cellPosx = 124;
  playerPtr->cellPosy = 376;
}
```

Notice that you only initialize the positions of two game players. This is for simplicity. There are sprites provided for only two different color warriors in the .BMP file. If you want to make four different color warriors for four players, just cut and paste the art and color it in to whatever color you want. Or you could use the art provided for your own 16-player game and just create a red team and a blue team. For instance, instead of checking for *playerNumber*, you could check for *teamNumber*, and so on. Whatever it is you choose to do, you must modify the code at this location to compensate for the changes.



Initializing Health and Armor

You have completed the logistics of the *PLAYER* initialization. The *PLAYER* node contains information about terrain map location as well as the sprites

required to draw the warrior. You're now going to add the warrior's attributes. The *Dungeon* example uses health, armor, and attack attributes for the warriors. One of the features of an RPG is that there are limitations on the characters' abilities. The warriors in *Dungeon* use these concepts as well. Listing 11-13 shows the code for initializing attributes.

LISTING 11-13 Initializing the PLAYER attributes

```
playerPtr->armor = 2;
playerPtr->maxArmor = 2;
playerPtr->health = 2;
playerPtr->maxHealth = 2;
playerPtr->attack = 2;
playerPtr->maxAttack = 2;
```

A warrior possesses the ability to pick up and carry objects. The objects in *Dungeon* include shields, swords, and potions. A warrior can carry up to eight items. When the warriors are initialized, they don't have any items in their possession. The objects are located on the map somewhere and must be retrieved.

If your game design interests are in RPG, you may want to expand these fields to include intelligence, speed, dexterity, and might. The *Dungeon* game can be the primordial ooze for your own RPG world. Now that you understand how your world is created, take a look at walking around in it.



Player Movement on the Terrain Map

The warrior moves in response to the commands of the game player on the local game instance. Input is provided through keyboard or gamepad input. A warrior may move forward or backward based on the current *facing* direction. A warrior's facing is controlled by turning the warrior to the left or right. As the warrior turns, its facing is changed to one of eight compass facing values: north, northeast, east, southeast, south, southwest, west, and northwest, as shown in Figure 11-11.

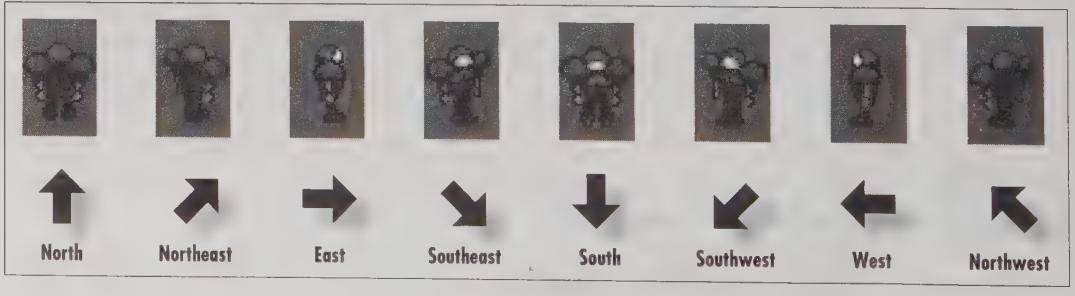


FIGURE 11-11 The compass facings of a dungeon warrior

Facings are set according to left and right rotations that I'll discuss shortly. When a facing is selected based on player input, the warrior can either stand and face in that direction, or move forward or backward. Facings are completely dependent on the player input.



Updating the Player Position

The player facing and movement is updated in the functions getKeyboard() and getJoystick(). However, the player position is not updated by these functions. The input routines can be considered as a means to set the warrior's direction settings. The terrain map and the local game's player are updated by playerMove(). Remember, this is a distributed computing environment; all the updates you make are to the local game's warrior. The updates are then sent out over the net to others participating in the game to update their terrain databases. The code to update the local game instance's player is provided in Listing 11-14.

LISTING 11-14 Moving the Player

```
void playerMove(int direction)
{

  int oldPosx;
  int oldPosy;
  int tempDirection;

  // Store current position just in case new movement cannot happen.
  oldPosx = playerCurrent->Posx;
  oldPosy = playerCurrent->Posy;
  processScreen = TRUE;
  tempDirection = playerCurrent->facing;

  // If the player is moving backwards then direction the player
  // is moving is opposite of the facing.  (Basically I didn't
  // want two switch statements, one for forward, and one for
  // backward movement).
  if (direction < 0)
    tempDirection = (tempDirection + 4) % 8;

  // The current position of the player is NULLed
  terrain[playerCurrent->Posy][playerCurrent->Posx].playerPtr = NULL;

  // Depending on current movement direction find new position.
  switch (tempDirection)
  {
    // Movement is Southwest.
    case FACE_SW:
      playerCurrent->Posx -= 1;
      playerCurrent->Posy += 1;
```

continued on next page

continued from previous page

```
      break;

    // Movement is South.
    case FACE_SOUTH:
      playerCurrent->Posy += 1;
      break;
    // Movement is Southeast.
    case FACE_SE:
      playerCurrent->Posy += 1;
      playerCurrent->Posx += 1;
      break;
    // Movement is West.
    case FACE_WEST:
      playerCurrent->Posx -= 1;
      break;
    // Movement is East
    case FACE_EAST:
      playerCurrent->Posx += 1;
      break;
    // Movement is Northwest.
    case FACE_NW:
      playerCurrent->Posx -= 1;
      playerCurrent->Posy -= 1;
      break;
    // Movement is North
    case FACE_NORTH:
      playerCurrent->Posy -= 1;
     break;
    // Movement is Northeast.
    case FACE_NE:
      playerCurrent->Posy -= 1;
      playerCurrent->Posx += 1;
      break;
    default:
      break;
  }
  // Check for terrain boundaries.
  if (playerCurrent->Posy > TERRAIN_HEIGHT)
    playerCurrent->Posy = TERRAIN_HEIGHT;
  if (playerCurrent->Posx < 0)
    playerCurrent->Posx = 0;
  if (playerCurrent->Posy < 0)
    playerCurrent->Posy = 0;
  if (playerCurrent->Posx > TERRAIN_WIDTH)
    playerCurrent->Posx = TERRAIN_WIDTH;

  // Check to see if player is trying to move into obstacle.
  int iPosx = playerPtr->Posx;
  int iPosy = playerPtr->Posy;

  if (wall_cells[terrain[iPosy][iPosx].cell].obstacle  == 3)
        || (terrain[iPosy][iPosx].playerPtr != NULL) )
```

```
  {
    // Restore old position.
    playerCurrent->Posx = oldPosx;
    playerCurrent->Posy = oldPosy;
    iPosx = oldPosx;
    iPosy = oldPosy;
    processScreen = FALSE;
  }
  // Have new terrain cell point to player
  terrain[iPosy][iPosx].playerPtr = playerCurrent;
  if(net)
    DGNPlayerPosNotify();
}
```

The *Dungeon* game uses the global variable *playerCurrent* to update the local host's player. In function doInit(), *playerCurrent* is assigned the address of *playerHead* after the list is initialized. The function playerMove() only updates the local host warrior referenced by *playerCurrent*. The process for moving a warrior involves saving the current position of the player. The *PLAYER* pointer in the current terrain map cell is set to NULL, indicating the player no longer occupies this space. If the move being made is invalid, such as out of boundaries, or into an obstacle, then you will have to restore the current valid position.

To determine direction, there are eight predefined facings in the *Dungeon* game. The first defined heading is FACE_NORTH with a value of 0. Moving in a clockwise rotation, the eight directions increment from 0 to 7. The compass direction tells you which direction the sprite is facing on the map. The facings are controlled by the user either through updates from DirectInput or the numeric keypad, as shown in Figure 11-12.

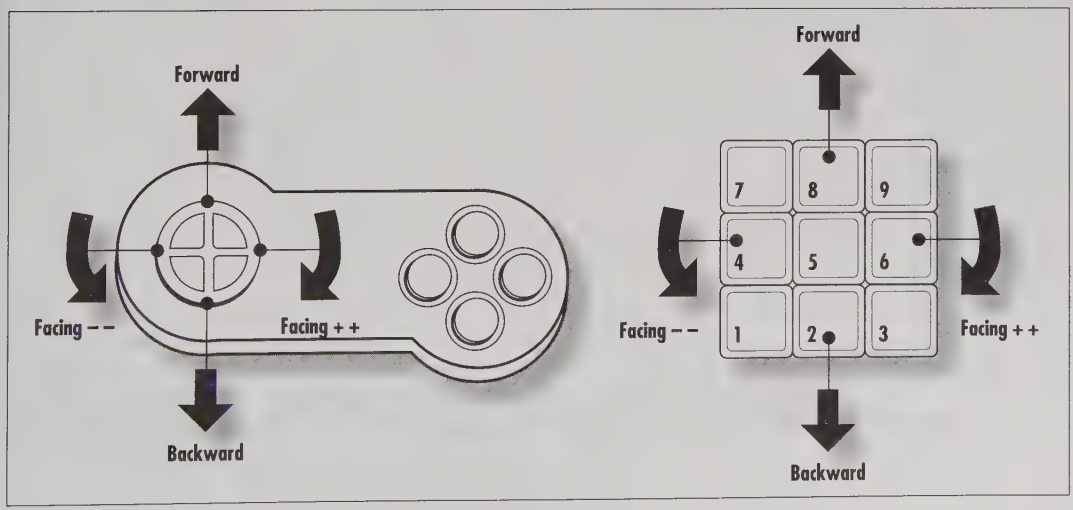


FIGURE 11-12 Warrior facings and user input

The facings provide the eight directions of motion for the warrior to walk around on the terrain map. The present direction is retrieved from *playerCurrent->facing* and passed into a switch statement. Based on the facing, the *currentPlayer->Posx* and *currentlyPlayer->Posy* fields are updated accordingly, as shown in Figure 11-13. The *Posx* and *Posy* values are the warrior's new position on the terrain map. The code for updating the *playerCurrent* position is shown in Listing 11-15.

LISTING 11-15 Updating the PLAYER's position

```
switch (tempDirection)
 {
   case FACE_SW:
     playerCurrent->Posx -= 1;
     playerCurrent->Posy += 1;
     break;
   case FACE_SOUTH:
     playerCurrent->Posy += 1;
     break;
   case FACE_SE:
     playerCurrent->Posy += 1;
     playerCurrent->Posx += 1;
     break;
   case FACE_WEST:
     playerCurrent->Posx -= 1;
     break;
   case FACE_EAST:
     playerCurrent->Posx += 1;
     break;
   case FACE_NW:
     playerCurrent->Posx -= 1;
     playerCurrent->Posy -= 1;
     break;
   case FACE_NORTH:
     playerCurrent->Posy -= 1;
    break;
   case FACE_NE:
     playerCurrent->Posy -= 1;
     playerCurrent->Posx += 1;
     break;
   default:
     break;
 }
```

The game player's movement must be checked to ensure that the warrior is being moved to a valid location on the terrain map. The warrior may not occupy a cell in the map that contains a wall or an obstacle. Most important, you must check to see that the warrior is not stepping out of the bounds of the terrain map. You first validate that the player is within the bounds of the map using the code in Listing 11-16.

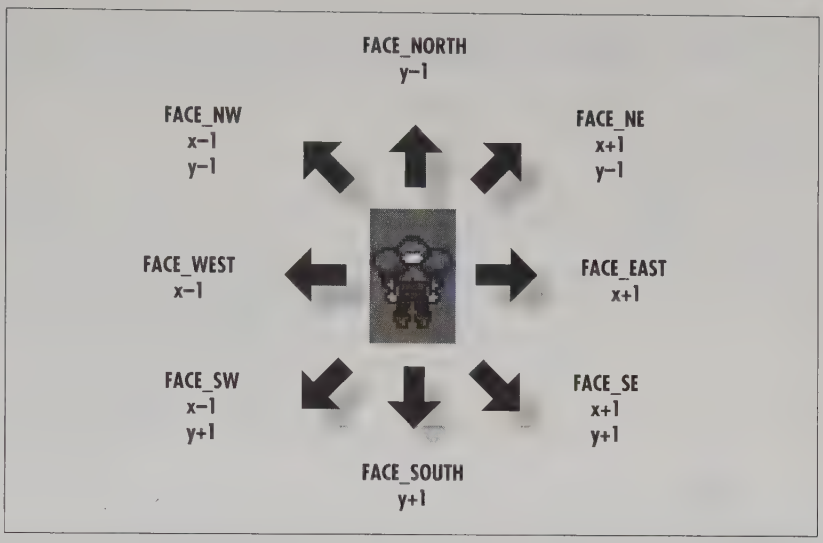


FIGURE 11-13

Terrain map coordinates and facings

LISTING 11-16 Checking the terrain map boundaries

```
if (playerCurrent->Posy > TERRAIN_HEIGHT)
  playerCurrent->Posy = TERRAIN_HEIGHT;
if (playerCurrent->Posx < 0)
  playerCurrent->Posx = 0;
if (playerCurrent->Posy < 0)
  playerCurrent->Posy = 0;
if (playerCurrent->Posx > TERRAIN_WIDTH)
  playerCurrent->Posx = TERRAIN_WIDTH;
```

What you're doing is constraining the player's movements to within the terrain map. If the *playerCurrent->Posy* is larger than the height of the terrain map, TERRAIN_HEIGHT, set the new *playerCurrent->Posy* to TERRAIN_HEIGHT. This gives the effect that the player is walking into an imaginary barrier, which is a visual indicator for the player to turn back into the terrain map bounds. Similar operations are performed for the other three (left, top, and right) boundaries of the map.

If the player's move is within boundaries, you must then check to see if the cell is available. Cells in the terrain map may have another warrior, wall, or other obstacle. The code to validate the new terrain map position is shown in Listing 11-17.

LISTING 11-17 Checking to see if the new cell is occupied

```
int iPosx = playerPtr->Posx;
int iPosy = playerPtr->Posy;

if ((wall_cells[terrain[iPosy][iPosx].cell].obstacle == 3)
```

continued on next page

continued from previous page

```
    || (terrain[iPosy][iPosx].playerPtr != NULL))
{
  playerCurrent->Posx = oldPosx;
  playerCurrent->Posy = oldPosy;
  iPosx = oldPosx;
  iPosy = oldPosy;
  processScreen = FALSE;
}
terrain[iPosy][iPosx].playerPtr = playerCurrent;
```

In the event that the new cell is occupied by an opponent or an obstacle, restore the original position from before the move. This simulates that the current heading is blocked or impassable. Otherwise, the new cell is valid and you move the warrior into this position. The terrain map cell *PLAYER* pointer is then updated to *playerCurrent*, indicating the cell is occupied.



Player Movement in the Distributed Environment

Once the local host accepts the valid move, the remote host(s) must also be updated. The last operation performed by function playerMove() is to send out the *playerCurrent* position on the net. The function that updates the remote players is DGNPlayerPosNotify(). All network communication utilizes DirectPlay messaging. The message structure for *Dungeon* is essentially a cut-and-paste rehack of your Chapter 10 *NetChess* example. The DirectPlay message structure for *Dungeon* is shown in Listing 11-18.

LISTING 11-18 The *Dungeon* DirectPlay messaging structure

```
typedef struct
{
  BYTE  msg;
  BYTE  posx;
  BYTE  posy;
  BYTE  armor;
  BYTE  attack;
  BYTE  health;
  BYTE  facing;
  DWORD dpid;
} NETDGNMSG, *LPNETDGNMSG;
```

A DirectPlay message is basically anything you want to define so long as it doesn't exceed 512 bytes in size. The message is sent out over the net using the function DGNPlayerPosNotify(), shown in Listing 11-19.

LISTING 11-19 Notifying remote nodes of PLAYER move

```
void DGNPlayerPosNotify()
{
  LPNETDGNMSG lpMsg;
  int msgLen;

  lpMsg = (LPNETDGNMSG) dpMsgBuf;
  lpMsg->msg = MSG_PLAYER_MOVE;
  lpMsg->posx = playerCurrent->Posx;
  lpMsg->posy = playerCurrent->Posy;
  lpMsg->armor = playerCurrent->armor;
  lpMsg->attack = playerCurrent->attack;
  lpMsg->health = playerCurrent->health;
  lpMsg->facing = playerCurrent->facing;
  msgLen = sizeof(NETDGNMSG);
  lpDP->Send( dpID, dpNodeID, 0, (LPSTR) dpMsgBuf, msgLen);
}
```

Again, for the sake of a simple example, the DirectPlay member Send() will only update one player whose node is *dpNodeID*. If you are designing a multiplayer dungeon of two or more players, you will need to construct a global array of DPID *dpNodeID[MAX_PLAYERS]*. To update all the players, you'll need to loop through all the *dpNodeID*s in the array. Figure 11-14 shows a loop calling Send() for a dungeon with four players. Note that you aren't using the standard client-server architecture as you did with *NetChess*. Any node can send a player move update at any time. The *playerCurrent* is relative to the local host. Any player not on the local host is an opposing game player. Therefore, each instance of the *Dungeon* game has a *playerCurrent* instance. This differs from *NetChess* where the server dictated who would send and when.

Once the message is sent out on the net, the remote game nodes listen for the update. The function in *Dungeon* to update a local host of a remote game player's move is DGNReadPosNotify(). The code for DGNReadPosNotify() is provided in Listing 11-20.

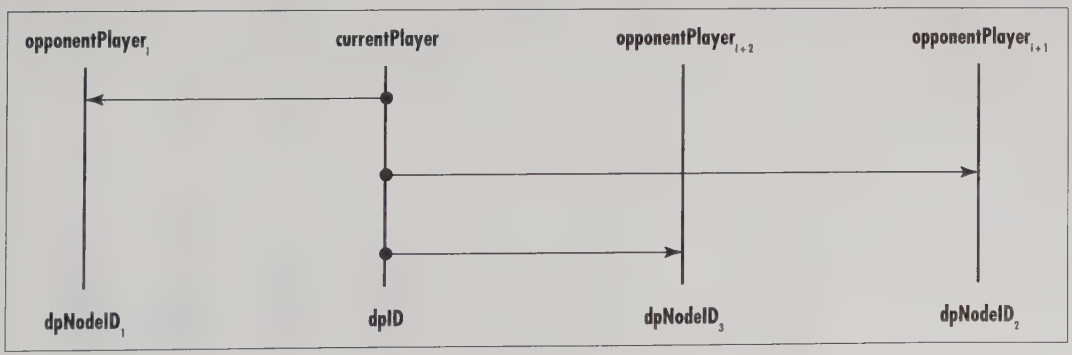


FIGURE 11-14 Sending a PLAYER move in a multiplayer dungeon

LISTING 11-20 Reading the PLAYER move update message

```
void DGNReadPosNotify()
{
  LPNETDGNMSG lpMsg;
  BYTE msg;

  lpMsg = (LPNETDGNMSG) dpMsgBuf;
  msg = lpMsg->msg;
  terrain[playerOpponent->Posy][playerOpponent->Posx].playerPtr
    = NULL;
  playerOpponent->Posx = lpMsg->posx;
  playerOpponent->Posy = lpMsg->posy;
  playerOpponent->armor = lpMsg->armor;
  playerOpponent->health = lpMsg->health;
  playerOpponent->attack = lpMsg->attack;
  playerOpponent->facing = lpMsg->facing;
  terrain[lpMsg->posy][lpMsg->posx].playerPtr = playerOpponent;
  processScreen = TRUE;
}
```

The function DGNReadPosNotify() only handles the single remote player scenario. If you were to expand the *PLAYER* structure for the list to include the DirectPlay data member DPID, you could update the list for a multiplayer (two or more players) game. This would involve grabbing the DPID in the *dpFrom* field of the DirectPlay member Read() and searching the list for this player, as shown in Figure 11-15. You would then update the player node with the message contents. Finally, the terrain map is updated with the new position of the warrior.

For a simple multiplayer dungeon, the message updates are trivial. The complexity increases exponentially as you increase the number of players to the game. For instance, you have discussed nothing about what would happen in the event of player collisions. That is, on the local node a warrior moves into a valid free cell. On a remote node another warrior moves into the identical valid free cell. It takes a finite amount of time for the message to go out over the net. Meanwhile, both warriors now occupy the same cell on the terrain map. Situations like these occur frequently in a distributed computing environment. As a developer, you must be aware of these conditions and handle them appropriately.

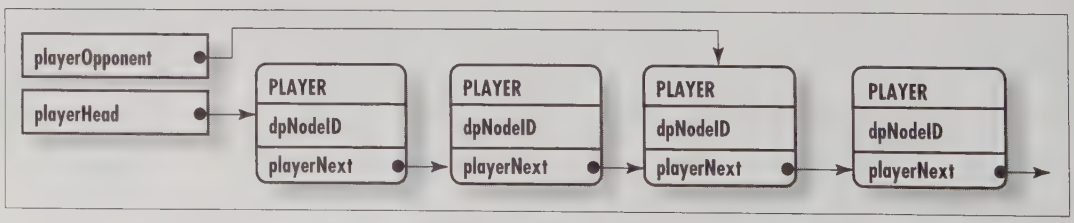


FIGURE 11-15 Updates in a multiplayer dungeon using DPID

Player Characteristics

The player attributes are what makes the gameplay interesting. For your game you may want to include a character selection screen for your multiplayer dungeon. Each character can have different strengths, vulnerabilities, dexterity, and cunning. This discussion will focus on how to use simple characteristics for building maps of undiscovered terrain, sneaking up on opponents, attacking opponents, and healing.



Player's Field of View

Giving players a limited field of vision provides a sense of realism to a game, as well as adding a new dimension to game strategy. The *Dungeon* game allows players to see in the direction they are facing, but objects and players behind the field of view are not visible. This ingredient heightens the gameplay by allowing players to sneak up and attack their opponents if they are not currently in their field of view. Potions and items are also included in the line-of-sight handling. Since these objects are to be found by the players, they are not readily visible unless the players have already discovered them or the item is in their line-of-sight.



The Bresenham Line Algorithm Revisited

To determine the player's line of site on the map terrain, you will cut and paste the Bresenham line-drawing algorithm from Chapter 6. In order to use the Bresenham algorithm, you must determine where your warrior is on the terrain map. The warrior will always be drawn to the display centered. Three conditions exist that you must handle. The first is to determine where the warrior is on the map. Second, close the region with a 15 x 15 cell-bounded rectangle, whose contents will be drawn to the screen. And third, determine what is visible and in the line of sight of the player in this bounded region.

Don't be confused by the 15 x 15 cell viewport. This is actually your 15 x 20 logical screen. The last five cells in each row of the display are reserved for a player status bar and a map viewer. This leaves you with a 15 x 15 screen to draw the terrain map cells to the display. For this example, your player is located dead center in the terrain map at location terrain[30][30]. You want to take a 15 x 15 cell snapshot of this region with the warrior located in the center as shown in Figure 11-16. You are going to use the variables *mapPosx* and *mapPosy* to identify the upper-left corner of your 15 x 15 cell snapshot. To center the warrior in the snapshot, calculate the *mapPosx* as *playerCurrent->Posx* - 7 (15/2 = 7.5 => 7) and *mapPosy* as *playerCurrent->Posy* - 7.

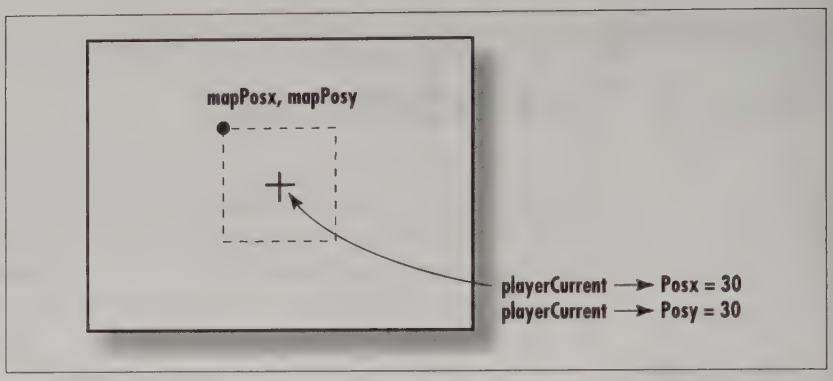


FIGURE 11-16
Creating a snapshot of the player's position

Once you have located your snapshot with the warrior centered in it, you are ready to determine what will be visible in the snapshot. The terrain map has a Boolean field, seen, for each cell on the map. All cells in the map are initialized with the seen field set to FALSE. You will use the Bresenham algorithm to draw an imaginary line from *playerCurrent->Posx* and *playerCurrent->Posy* to the cell *mapx* + i and *mapy* + *j*. The imaginary line is drawn in the direction you are facing. The function lineOfSight() is your recycled code from the original function Bresenham line algorithm in Chapter 6. All the cells to be drawn have a visibility factor of 0 to 3. The visibility field for each cell uses 0 for a transparent cell, meaning you can see things on the other side of the cell. A visibility value of 3 means the cell contains a solid wall so you can't see what's on the other side of the wall. The function drawCells() will not draw a tile beyond this boundary unless the cells on the other side of the wall have already been seen, as shown in Figure 11-17.

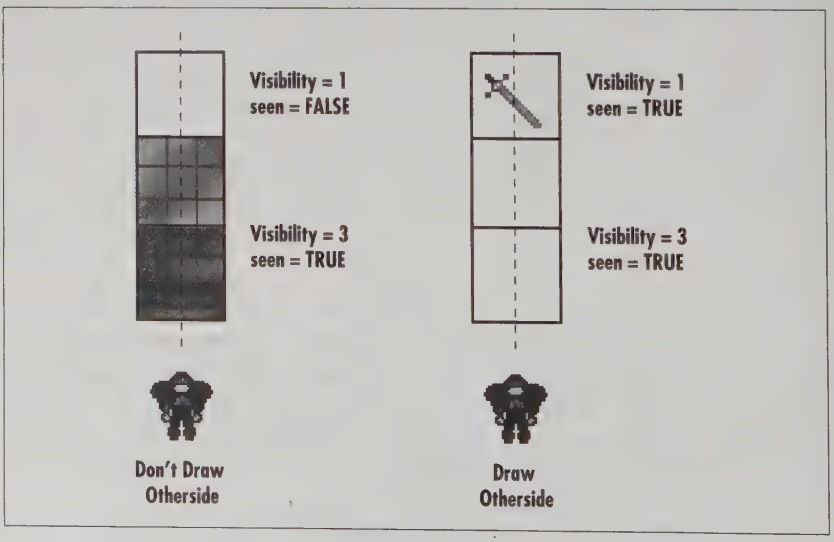


FIGURE 11-17
Drawing the other side of a solid wall

Any object that is beyond a solid wall will not be drawn unless the player had previously been on the other side of the wall to set the seen flag to TRUE. The code for function lineOfSight() is shown in Listing 11-21. What you're doing with the Bresenham algorithm is using it for a simple raycaster. You draw a line to a cell and look up the cell's visibility.

LISTING 11-21 Using lines to determine visibility

```
int lineOfSight(UINT x1, UINT y1, UINT x2, UINT y2)
{
  int      dx, dy;
  int      step;      // direction to proceed
  int      clearLine;
  UINT     sourcex;
  UINT     sourcey;
  UINT     destx;
  UINT     desty;

  sourcex = x1;
  sourcey = y1;
  destx = x2;
  desty = y2;
  clearLine = 0;
  // if y1 > y2 swap coordinates
  if (y1 > y2)
  {
    swapTermsInt(&y1, &y2);
    swapTermsInt(&x1, &x2);
  }

  // calculate deltas
  dx = x2 - x1;
  dy = y2 - y1;

  if (dx > 0)
  {
    step = 1; // proceed in the positive direction
  }
  else
  {
    dx = -dx;
    step = -1;
  }

  if (dx > dy)
    clearLine = ApproxNextY( x1, y1, dx, dy, step,
                  sourcex, sourcey, destx, desty);
  else
    clearLine = ApproxNextX( x1, y1, dx, dy, step,
                  sourcex, sourcey, destx, desty);

  return(clearLine);
}
```

The function lineOfSight() is called from function DrawCells() in the *Dungeon* program. The portion of the terrain map you want to draw to the display, your snapshot, is bounded by the rectangle *mapPosx*, *mapPosy*, *mapPosx + 15*, *mapPosy + 15*. When you draw each cell in this region using a nested for *i* and for *j* loop, you will call lineOfSite() to see if the object you are drawing is visible, or if its visibility is being blocked by a solid wall, in which case the cell will only be drawn to the screen if the cell's seen flag is TRUE. The code for drawCells() is provided in Listing 11-22.

LISTING 11-22 Drawing cells using the Bresenham algorithm

```
void drawCells( void )
{

  PLAYER              *pwrTemp;
  ITEM                *itemTemp;
  int                 i,j;
  int                 left;
  int                 down;
  int                 tempdown;
  int                 cellnum;
  RECT                rcRect;
  UINT                LOSModifier;

  // Try to determine the terrain cell that will be in the
  // top left corner.
  // Subtract 7 since there should be 7 cells in the Y
  // negative direction above the center of the screen
  mapPosy = playerCurrent->Posy - 7;
  // Since 15 cells can be displayed in the Y direction,
  // we must make sure the screen does not go out of bounds.
  if (mapPosy > TERRAIN_HEIGHT-15)
    mapPosy = TERRAIN_HEIGHT-15;
  if (mapPosy < 0)
    mapPosy = 0;

  // Subtract 7 since there should be 7 cells in the X
  // negative direction from the center of the screen.
  mapPosx = playerCurrent->Posx - 7;
  // Since 15 cells can be displayed in the X direction,
  // we must make sure the screen does not go out of bounds.
  if (mapPosx < 0)
    mapPosx = 0;
  if (mapPosx > TERRAIN_WIDTH-16)
    mapPosx = TERRAIN_WIDTH-16;

  // The actual Y offset where to put the current row of cells.
  down = 60;
  left = 0;
  cellnum = 16;
  // Outer loop of drawing cells.
```

```
for (j = 0; j < 15; j++)
{
  for (i = 0; i < cellnum; i++)
  {
    // Now reference the terrain
    // array and get what kind of cell it
    // holds then take that information
    // and reference the cells array
    // to get the source rectangle.
    indx = i + mapPosx;
    jndx = j + mapPosy;
    rcRect.left =
         wall_cells[terrain[jndx][indx].cell].posx;
    rcRect.top = wall_cells[terrain[jndx][indx].cell].posy;
    rcRect.right = wall_cells[terrain[jndx][indx].cell].posx
           + wall_cells[terrain[jndx][indx].cell].width;
    rcRect.bottom = wall_cells[terrain[jndx][indx].cell].posy
           + wall_cells[terrain[jndx][indx].cell].height;
    tempdown = down - wall_cells[terrain[jndx][indx].cell].height;
    if (tempdown < 0)
    {
      rcRect.top = wall_cells[1].posy - tempdown;
      tempdown = 0;
    }

    if ((facingDirection(playerCurrent->Posx,
           playerCurrent->Posy, indx,jndx,
           playerCurrent->facing)
           && (LOSModifier = lineOfSight(playerCurrent->Posx,
           playerCurrent->Posy, indx, jndx)) < 3))
    {
      terrain[jndx][indx].seen = TRUE;
      // Now draw the cell.
      drawOneToBack(left, tempdown, rcRect, TRUE);

      if (((itemTemp = terrain[jndx][indx].itemPtr) != NULL))
      {
        rcRect.left = item_cells[itemTemp->type].posx;
        rcRect.top = item_cells[itemTemp->type].posy;
        rcRect.right = item_cells[itemTemp->type].posx
           + item_cells[itemTemp->type].width;
        rcRect.bottom = item_cells[itemTemp->type].posy
           + item_cells[itemTemp->type].height;
        tempdown = down - item_cells[itemTemp->type].height;
        if (tempdown < 0)
        {
          rcRect.top = item_cells[itemTemp->type].posy
              - tempdown;
          tempdown = 0;
        }
        drawOneToBack(left, tempdown, rcRect, FALSE);
      }
```

continued on next page

continued from previous page

```
  if ((pwrTemp=terrain[jndx][indx].playerPtr)!= NULL)
  {
    //Draw player.
    rcRect.left = pwrTemp->cellPosx+1;
    // If player is attacking then draw different cell
    if (pwrTemp->firing)
      rcRect.left += 93;
    // Take into account facing of player
    rcRect.top = pwrTemp->cellPosy + (pwrTemp->facing * 47)+1;
    rcRect.right = rcRect.left + 31-2;
    rcRect.bottom = rcRect.top + 47-2;
    // Draw the player
    tempdown = down - 47;
    drawOneToBack(left,tempdown, rcRect, FALSE);
  }   // if player in cell.
}   // if cell is visible
else
{
  if (terrain[jndx][indx].seen)
  {
    drawOneToBack(left, tempdown, rcRect, FALSE);
  }
  else
  {
    // draw non-visible cell.
    rcRect.left = wall_cells[0].posx;
    rcRect.top = wall_cells[0].posy;
    rcRect.right = wall_cells[0].posx + wall_cells[0].width;
    rcRect.bottom = wall_cells[0].posy + wall_cells[0].height;
    tempdown = down - wall_cells[0].height;
    if (tempdown < 0)
    {
      rcRect.top = wall_cells[0].posy - tempdown;
      tempdown = 0;
    }
    // Now draw the cell.
    drawOneToBack(left, tempdown, rcRect, FALSE);
  }
}  // else cell is not visible
if ((terrain[jndx][indx].playerPtr) == playerCurrent)
{
  //Draw player
  rcRect.left = playerCurrent->cellPosx+1;
  // If player is attacking then draw different cell
  if (playerCurrent->firing)
    rcRect.left += 93;
  // Take into account facing of player.
  rcRect.top = playerCurrent->cellPosy
        + (playerCurrent->facing * 47)+1;
  rcRect.right = rcRect.left + 31-2;
  rcRect.bottom = rcRect.top + 47-2;
  // Draw the player
```

```
            tempdown = down - 47;
            drawOneToBack(left,tempdown, rcRect, FALSE);
          }   // if player in cell.
          left += 30;
        }   // for (i = 0...
        down += 30;
        left = 0;
      }   // for (j = 0...
}
```

The body of the function looks pretty long for drawCells() but don't let this deceive you. Break the code down into what it is doing. The main body of the code is the nested loops to walk through the cells you want to display. Which cells are drawn to the snapshot is determined by the *huge* if statement inside the nested loop. Using the player facing and its line of sight, you either draw the cell, object, and opponent player, or you don't.



Player's Health

The premise of an RPG game is that the last warrior or group of warriors standing wins. The idea in these games is to conquer and to inflict damage on opposing warriors. RPG games use a roll of the dice to determine how much damage a warrior endures when being attacked. For your own game designs you will want to design your own dice emulator. Dice in RPGs are as unique as each game itself. Some dice are polyhedrons and not the typical six-sided dice. This demonstration will use a simple random number generator to determine damage on opponents when they are under attack.



Attacking the Opponent

To attack your opponent in *Dungeon* you must be in a cell that borders your opponent's cell. The attack also requires that your player is facing the opponent's cell, using one of the eight compass directions. By using the imaginary grid in Figure 11-18, you can determine if an attack will occur. You see the *currentPlayer* located in the center of the grid with the facing set to FACE_NE. In the grid northeast of the *currentPlayer* is the *opponentPlayer*. The cells are adjacent, so an attack may take place. The opponent does not have to be facing you to start the assault. But the *currentPlayer* must be facing the *opponentPlayer*.

The function attackPlayer() is called when the user presses the game pad button 1 or the [A] key on the keyboard, shown in Figure 11-19. If the attack conditions are valid, the *opponentPlayer* is on a bordering cell and the *currentPlayer* is facing the cell, the attack will commence. You'll see in Listing 11-23 that the damage inflicted is randomly generated.

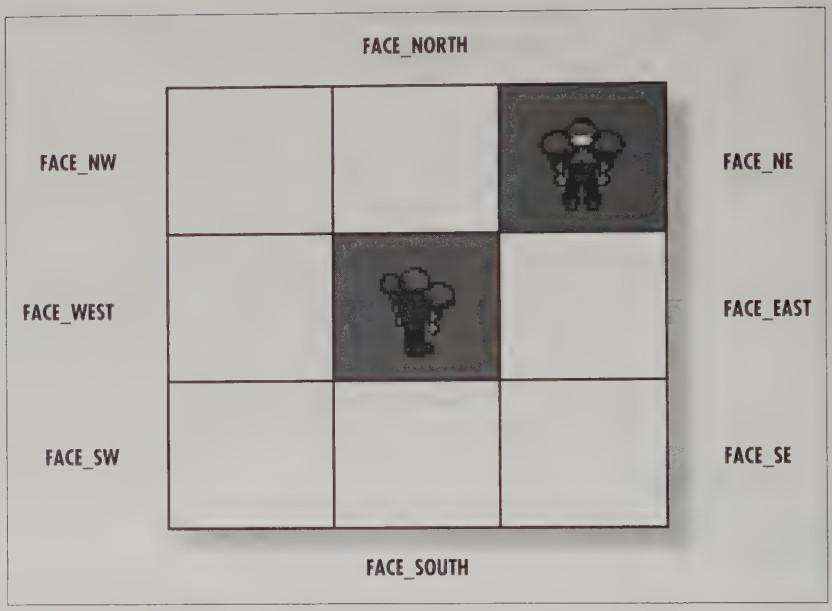


FIGURE 11-18
Locating the *opponentPlayer* for the attack

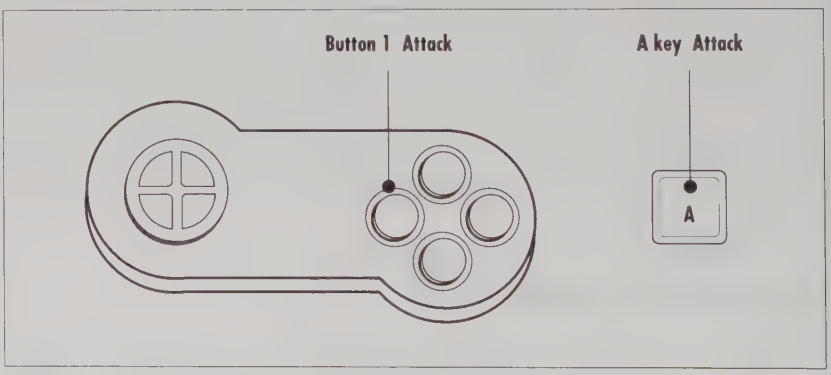


FIGURE 11-19
The input controls for attacking

LISTING 11-23 Attacking the opponent player

```
BOOL attackPlayer()
{
  PLAYER *playerPtr;
  POINT location;
  int damage;

  switch(playerCurrent->facing)
  {
    case FACE_NORTH:
      location.y -= 1;
      break;
```

```
    case FACE_NE:
      location.x += 1;
      location.y -= 1;
      break;
    case FACE_EAST:
      location.x += 1;
      break;
    case FACE_SE:
      location.y += 1;
      location.x += 1;
      break;
    case FACE_SOUTH:
      location.y += 1;
      break;
    case FACE_SW:
      location.x -= 1;
      location.y += 1;
      break;
    case FACE_WEST:
      location.x -= 1;
      break;
    case FACE_NW:
      location.x -= 1;
      location.y -= 1;
      break;
  }
  // check to see if the opponent is in that location
  if (terrain[location.y][location.x].playerPtr != playerOpponent)
    return FALSE;
  if (net)
    DGNPlayerHitNotify( damage );
  return TRUE;
}
```

The *opponentPlayer* is a remote player. So while you're giving them a good pounding on your local host, you must update the remote player. The function DGNPlayerHitNotify() sends a message to the remote player to inform him or her of your inhospitable nature. But be careful of what you start; the *opponentPlayer* may have a few potions up his or her sleeve.



Using Potions

Scattered on the *Dungeon* terrain map are swords, shields, and potions. Each cell in the terrain map has a pointer to the ITEM list. If pointer *terrain[x][y].itemPtr* is NULL, there is no item in that cell of the map. Visually, you can see the item if one exists, because it will be drawn to the display in that location. The item may be picked up by maneuvering the warrior into that cell and pressing button 2 on the gamepad or the letter P on the keyboard, as shown in Figure 11-20.

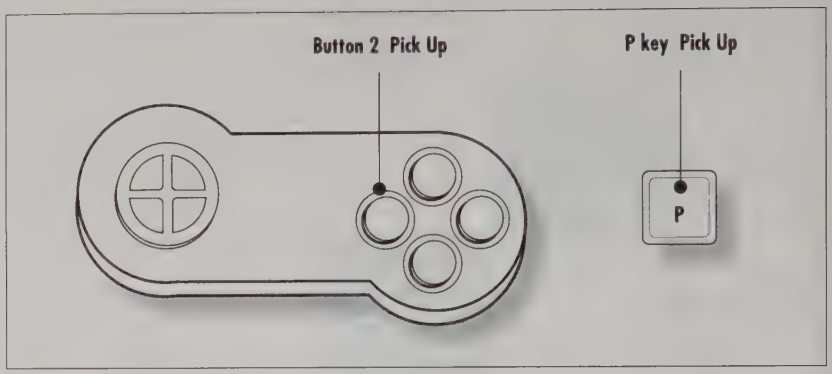


FIGURE 11-20
The input controls for picking up items

A player may pick up as many as eight items. When the player attempts to pick up an item a check is made to see how many items the player currently has. If he or she has eight items, then the object is left alone on the map. Otherwise, the item is retrieved and the pointer in the cell to the item is assigned to NULL. The function pickUpItem(), in Listing 11-24, is used to retrieve items from the terrain map.

LISTING 11-24 Picking up items on the map

```
BOOL pickUpItem(void)
{
  ITEM *tempPtr;
  int i;
  int iPosx = playerPtr->Posx;
  int iPosy = playerPtr->Posy;

  // return if no item in the cell
  if ((tempPtr
        = terrain[iPosy][iPosx].itemPtr)
        == NULL)
    return FALSE;

  i = 0;
  // item is in square
  // check to see if item is one of the basic item types
  // shield, attack, health
  if (itemPtr->subtype == 0)
  {
    switch(tempPtr->type)
    {
      case 0:
        playerCurrent->attack += 1;
        break;
      case 1:
        playerCurrent->armor += 1;
        break;
```

```
      case 2:
        playerCurrent->health += 1;
        break;
    }
    terrain[iPosy][iPosx].itemPtr = NULL;
    if (net)
      DGNPlayerGetNotify();
    return TRUE;
  }
  else
  {
    // item is in the square. search item list till end
    while(playerCurrent->itemsCarried[i] != NULL & i < 8)
      i += 1;
     // if i == 8, player already holds 8 items
     if (i == 8)
    return FALSE;
    playerCurrent->itemsCarried[i] =
       terrain[iPosY][iPosx].itemPtr;
    terrain[iPosY][iPosx].itemPtr = NULL;
    if (net)
      DGNPlayerGetNotify();
    return TRUE;
  }
}
```



Selecting an Item to Use

There are currently three simple potions the player may use in *Dungeon*. These potions affect the *currentPlayer*'s agility. For instance, a potion can speed up, slow down, or paralyze the *currentPlayer*. Players won't know which potion does what until they use it. A potion is selected from the right side of the display when the cursor turns into the Egyptian eye. Placing the cursor over the desired potion and pressing the left mouse button will select it. The code for selecting a potion is shown in Listing 11-25.

LISTING 11-25 Selecting a potion using the mouse pointer

```
case WM_LBUTTONDOWN:
  if (Mouse.xpos < 480)
  {
    if (!net)
    {
      changeSquare();
      processScreen = TRUE;
      Mouse.initial[0] = Mouse.initial[1] = TRUE;
    }
  }
```

continued on next page

continued from previous page

```
  if (Mouse.xpos > 480 && Mouse.ypos > 160 && Mouse.ypos < 224)
  {
    setTimeDelay(TIME_USE_ITEM);
    useItem(Mouse.xpos, Mouse.ypos);
    processScreen = TRUE;
  }
  break;
```

The function useItem() is called when the message WM_LBUTTONDOWN is processed. The current coordinates in *Dungeon*'s Mouse structure are passed in as calling arguments. The mouse coordinates are checked to see which potion was selected by the player and to validate if a potion was even selected at all. You then loop through the items you are carrying to locate the potion you selected from the screen. Once the item is located in the possession of the *currentPlayer*, the item is passed to a switch statement to activate the potion. All three potions used here affect the player's agility. You may want to modify this to turn your opponent into a newt. The source for using potions is shown in Listing 11- 26.

LISTING 11-26 Using potions in *Dungeon*

```
void useItem(int posx, int posy)
{
  int     position, counter, itemNumber;
  ITEM    *itemPtr;

  // Find out which position was selected from screen.
  position = (posx - 490)/32;
  if (posy > 192)
    position += 4;

  counter = 0;
  itemNumber = -1;
  // Now check to see if an item actually shows at that spot.
  while(itemNumber != position && counter < 8)
  {
    if (playerCurrent->itemsCarried[counter] != NULL)
      itemNumber += 1;
    counter += 1;
  }

  // If counter = 8 then item was not found.
  if (counter == 8)
    return;

  counter -= 1;

  // counter holds offset to item that user wants to use.
  itemPtr = playerCurrent->itemsCarried[counter];
  switch ( itemPtr->type)
  {
```

```
    case 3:
      playerCurrent->timeModifier = (float)PLAYER_MOD_FAST;
      playerCurrent->stopTimeModifier =
                GetTickCount() + TIME_MOVE_MODIFIER;
      break;
    case 4:
      playerCurrent->timeModifier = (float) PLAYER_MOD_SLOW;
      playerCurrent->stopTimeModifier =
                GetTickCount() + TIME_MOVE_MODIFIER;
      break;
    case 5:
      timeDelay = TIME_PARALYZE;
      break;
    default:
      break;
  }
  playerCurrent->itemsCarried[counter] = NULL;
}
```

A nice addition to the potions would be mines or other weapons. You could add some art for a floor tile with a bump to simulate a mine. Time bombs would be a neat addition too. Imagine the *opponentPlayer* accidentally using the paralyze potion on himself or herself, and you could whip out the ol' time bomb and split.



Conclusion

The examples provided here for *Dungeon* are intentionally left simple. There are limitless possibilities in which you can modify this code or just write your own RPG. The concepts covered here illustrate how to incur damage and use objects such as potions to heighten the gameplay. Many of you probably have your own volumes of RPGs you designed. Now you can cut and paste the *Dungeon* code to bring your own creations to life.

Keep in mind some of the concepts I mentioned for multiplayer dungeons. Create the linked list of players and then create an array of pointers, *opponentPlayer[MAX_PLAYERS]*. This will allow you to host multiple players in the *Dungeon* game. Passing the *dpNodeID* along in the message will help decipher from which player the message came. Once you get this working you can start your own dial-in virtual reality service.

12

AGAMEMNON'S CONCISE GUIDE TO VIRTUAL REALITIES



AGAMEMNON'S CONCISE GUIDE TO VIRTUAL REALITIES

Now do you not see that the eye embraces the beauty of the world? It is the lord of astronomy and the maker of cosmography; it counsels and corrects all the arts of mankind; it moves men to different parts of the world; it is the prince of mathematics, its sciences are certain; it has measured the heights and sizes of stars, it has found the elements and their locations...has generated architecture, perspective, and the divine art of painting.

Leonardo da Vinci (1452–1519)

Through the ages, mankind has feverishly attempted to heighten its state of being and fashion its own reality. The Ancient Egyptians, who had a culture of art, mysticism, and to some extent, hidden knowledge, constructed large pyramids to act as vessels to take their royalty to the afterlife. The architecture and means of constructing these structures are still a mystery shrouded by the art of magic. Every society in human history has left a legacy of shaping its surroundings into what the people believed to be a better existence. Some examples would be the great Inca civilizations in the jungles of Peru, or the mythological architectures in Greece and Pompeii. Writing, art, and architecture have focused on lifting people from their present state of being to a new higher ground. In the European Renaissance, artists, mathematicians, and architects used their skills to create a heavenly reality within the walls of large cathedrals. Because of the use of geometry and projection, people entering these large ornate structures would immediately feel the sense of being removed from their existence into the presence of their saints and of God.

The architects of the Renaissance, unlike their Egyptian counterparts, are known to us. There was Leonardo da Vinci, Michelangelo, Peruzzi, Botticelli, and many others. These people and countless others through the ages are also the architects of modern day virtual reality.

This chapter will look at techniques used by PC game designers to create virtual realities. These aspects will include a discussion on how to set Mode X, a popular video mode for such games, polygons, clipping and filling, fixed-point math, and an example of binary-space partitioning.

EPISODE GROUND ZERO



Returning to my body was a strange sensation. But who's to say I ever left it? Even I can't answer that question. I just know the moment prior to when I regained feeling in my body, I had the sensation of falling, such as in a dream when just before the moment of impact your body jerks and you awaken realizing you were in bed and it was only a dream. However, this was no dream.

I pushed myself out of 13's fascinating creation. I wanted nothing more to do with his device. I had spent an indeterminate amount of time in virtual space and I didn't know if my body had regained its sea legs. Yet with my sudden return from the other side, I knew we weren't out of hot water just yet.

Carol, or Zero, which was what Carol had become, set the artifact in a self-destruct mode. The artifact had tentacled its way into numerous power sources in this facility and was drawing a substantial load for its own demise. The ground beneath us trembled as the artifact surged in its overload.

13 came over and helped me to my feet. Old ceiling tiles and loose earth started to fall.

"Where's Kala?" I yelled over the rumbling. "We don't have any time; we're leaving now!"

"She's not doing so well," 13 yelled over the violent tremors. "While you were in the artifact, the Grinch captured her and tried to interrogate her using inhumane truth agents. I managed to recover her using a service robot. She's still unconscious." 13 pointed to her lying on some blankets over in a corner.

"Let's move. We'll both have to carry her out," I commanded.

This was going to be an interesting feat. My head was dizzy and I didn't feel one hundred percent, but we had no other choice. I had to walk, and I had to help carry Kala. I reached down and felt a head-rush like I was going to pass out. I fought the sensation and bent down to my knees to help lift Kala.

"Do you have her, 13?" I grunted.

"I have her," he replied.

"Ready? On the count of three. One...two...three!" I could feel my teeth grind as I mustered the energy and will to stand up with the additional weight.

Kala's face hung down limp, like no muscles were in her body to hold her head up. Her face was beaded with a sticky sweat. "Come on Kala, fight it. We're getting out of here," I muttered to myself.

Kala suddenly tried to lift her head. It bobbed uncontrollably and then she looked at me. "When I was nine, I wanted a Spirograph for Christmas, but I didn't get one." Kala spoke insensibly; and then her head fell down again.

"Come on, girl, hold on just a little longer," I tried to reassure her.

"Go to the right. There is an access tunnel to another military installation in there," 13 instructed.

"We're going to need quite a bit of distance, 13. Agamemnon has knowledge of critical mass. I saw two shielded hemispheres inside the artifact. When those two pieces come together, we have an instant atomic bomb," I informed.

"Don't worry," replied 13. "I have it all covered!"

When we came to the junction 13 had led us to, there was a control room that was a dead-end. The ground beneath us began to shake even more violently. I helped support Kala against a wall as she rambled openly about a biology teacher she hated in high school.

13 worked quickly and threw some levers and toggle switches on a control panel. From the look of the panels, these controls probably hadn't been operated since the late sixties.

"This room was the control center for classified nuclear rocket engine testing. There is a special escape vehicle here that was to be used in the event one of these devices had a mishap," 13 informed.

He threw some more switches, a side wall opened revealing what looked like a rifle bullet large enough for people to get into. The front of the bullet tapered off into wall and wasn't exposed.

"In a few seconds we'll be halfway to New Mexico. With any luck we'll make it in one piece," 13 added. "You and Kala get in. I'll initialize the launch sequence."

I did as 13 instructed and put Kala into the escape pod. The interior was cumbersome and hard to move around in. I strapped Kala in and climbed out to see if 13 needed any help. He was already right behind me and clambered his way into the pod and strapped himself in. The countdown had commenced. I reached up to close the door but it wouldn't shut. I tried tugging on it again, but it was stuck open. The old clock dial ticked in seconds to three, then to two. I looked up and saw what the problem was.

The possessed Grinch, or Agamemnon, held the pod door open. He had found a gun in his pursuit of me and took aim at my forehead. I knew this was the end of me for sure. The ground shook ever more violently as the artifact ticked down in its self-destruct. For a second I thought that Agamemnon squeezed the trigger. My head jerked, but it wasn't a gunshot wound, it was the kick of the escape pod rocket engines igniting. The rapid acceleration tore the open door off as we raced down the tunnel. There was a very violent jerk, then a blinding flash as the artifact rapidly assembled its critical mass.

In an instant it was all over. The hidden secrets of the artifact and secret base were no more. My fillings were being violently rattled from my teeth as the

rocket sled sped away. Kala giggled next to me, no doubt the work of the drugs circulating in her body. We weren't vaporized, and there's no telling what happened to the Grinch. It would be impossible to assume Agamemnon survived that, but the sorcerer was ingenious. I could only hope this was his demise.

As for the three of us, we have a new beginning and new identities. I was killed in a shark attack on the ten o'clock news, 13 never existed, and I doubt the Grinch left any traces of Kala's past life. Carol downloaded all of Agamemnon's data into my head. I knew secret bank accounts and how to play the stock market. I don't think we'll ever have to work a day in our lives again.



Mode X: The Medium of the Mystics

As each Pentium processor becomes increasingly powerful over its predecessor, without a doubt, the next popular screen resolution for games will soon be 640 x 480 pixels using the Windows 95 Direct X Game SDK. However, the many popular games, such as all the *DOOM* games, *Heretic, Hexen,* and other highly marketable games, all use 320 x 200 and 320 x 240 resolution video Mode X. Do not confuse the 320 x 200 Mode X with the 320 x 200 resolution Mode 13h, popular with shareware developers. The 320 x 200 Mode X is popular with game developers and supports page flipping. This video mode is also ideal for games that require intense 3D projections and computations such as *DOOM* and *Hexen*, where each screen pixel must be written individually for texture mapping. The Direct X API and fast Pentium processors will help break this barrier and allow developers to work with 640 x 480 screen resolution modes for the next generation of virtual reality games. However, there is still considerable cost in migrating popular games to new platforms. Art for games can sometimes be the single most expensive portion of game development. A lot of games being ported to Windows 95 will still utilize art for 320 x 200 and 320 x 240 Mode X screen modes. *DOOM* and *Hexen* for Windows 95 are classic examples of this migration. Other games such as *Thexder* and *Mortal Kombat III* have also followed. This section will look at how to initialize DirectDraw to set up Mode X screens with up to three flipping surfaces. For those developers looking for a quick and easy way to rapidly prototype or migrate to a Direct X game using your current art, this discussion will be of great interest.



Video Mode X

Video Mode X is extremely popular with DOS developers. Most of the developers I worked with in collecting material for this chapter were working

on their first Direct X games for Windows 95. The common response from developers was that they were performing a direct port of an already existing DOS game. For this reason I decided to rehash my DDTOOLS library and produce a new cut that is exclusively for supporting Mode X games using DirectDraw. The new library, which I call DXMODEX for DirectDraw Mode X, will be presented here in its entirety.



Which Mode X?

As a Doctor Dobb's reader, when someone mentions video Mode X, I instinctively think of the Michael Abrash articles on video Mode X, which is 320 x 240 pixels in resolution. However, in the game industry Mode X is distinctively 320 x 200 pixels in resolution with supported page flipping. The Mode X library I will demonstrate will support both resolutions with some added twists using the Microsoft Direct X Game SDK. For instance, DirectDraw interface hooked to supported video hardware will allow use of triple buffer page flipping. This technique is currently used in *DOOM 95* using DirectDraw.

The good news about Mode X is that it is supported by the Direct X libraries; the bad news about Mode X, using DirectDraw, is that it can be as complex and mysterious as its DOS video counterpart. When I approached developers with problems I encountered, I either received a devilish grin, from those who were successful, or outbursts of rage from those who were still fumbling with this video mode. Why is Mode X in Direct X so difficult? Well, actually, it isn't. The difficulty lies in a statement I made way back in Chapter 1, *"Let go of your old DOS coding habits."* Well, I'm officially a hypocrite.



DirectDraw Mode X and Video Card Memory Architectures

The mistake I, and others, made when using DirectDraw using display Mode X was that we assumed DirectDraw would work miraculously as it had when we were using a 640 x 480 display mode with a depth of 8 bpp and one back surface for flipping. 320 x 200 and 320 x 240 display modes must be conscientious about video memory architecture. The nature of all the confusion is that DirectDraw is a video memory manager. The memory is presented as a flat model, so programmers can treat address space as a logical rectangular configuration. The idea is that if I have a 640 x 480, 8 bpp, double-buffered surface, I can address consecutive rows of memory in the surface at 640 byte offsets. In video memory, the offset to the next scan line is called pitch. A typical 1M video card has two popular memory architectures, as shown in Figure 12-1. The first example in the figure shows a video card with a pitch of 640 bytes. The DirectDraw primary surface is 640 x 480 pixels and has a width of 640 bytes. The second example in Figure 12-1 shows a video card whose pitch is 1024 bytes, and the DirectDraw primary surface is 640 bytes wide.

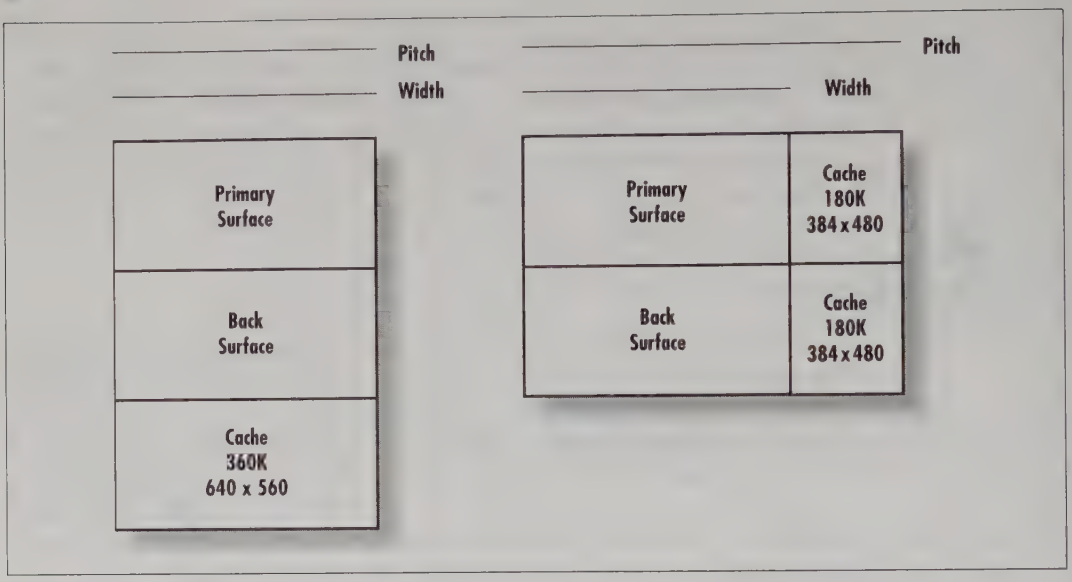


FIGURE 12-1 Memory configurations for 1M video cards

The lesson to be learned here is that when writing to a DirectDraw surface, use the video card's pitch to calculate offsets and not the DirectDraw surface's width. The DirectDraw surface width is how many bytes of the card's scanline are owned by a DirectDrawSurface. The pitch is the value that must be added to the start of a scanline to get to the start of the next line in the DirectDraw surface. Consider, for example, that you are copying a 640 x 480, 8 bpp, DIB image into a DirectDraw surface. The DIB image is referenced by the pointer, *lpdibBits;* the next row of image bits in this buffer is at an offset of 640 bytes. The destination you are writing the image bits to is a DirectDrawSurface. The pointer, *lpsurfBits,* points to the buffer returned from the Lock() call. The next row in the surface is at an offset of ddsd.lPitch, where the data structure ddsd is a DIRECTDRAWSURFACEDESC structure. The simple example for using pitch in the line offset calculations is shown in Listing 12-1.

LISTING 12-1 Using pitch to calculate DirectDrawSurface offsets

```
int i;
for (i=0; i<480; i++)
{
  memcpy(lpsurfBits, lpdibBits, 640);
  lpsurfBits += ddsd.lPitch;
  lpdibBits += 640;
}
```

Another video card architecture, which is determined to be a leading cause of programmer substance abuse, is bank-switched memory. When you create

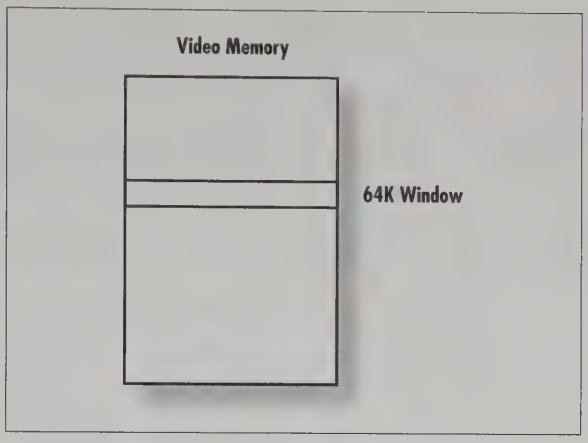


FIGURE 12-2
Bank-switched memory

your primary DirectDrawSurface, use the DirectDrawSurface member GetCaps() to check if the device is bank-switched. Perform a logical AND using the flag DDSCAPS_BANKSWITCHED. Why is this important and how does it keep you out of the Betty Ford programmer's clinic? Bank-switched memory is an architecture that presents its memory in 64K windows; see Figure 12-2. Recall back in Chapter 2, when the DIB bits were unpacked so they were DWORD-aligned? Bank-switched memory takes advantage of DWORD-aligned writes and is very fast. Because of this precedence, it is used in some fast video cards, such as the S3 chipset. In a bank-switched architecture, if the data isn't DWORD-aligned, the image bits can be scattered across 64K boundaries. For those of you squeezing frame-rates out of your Direct X applications, this means DirectDraw will be wasting time between page faults, flopping back and forth between boundaries.

How do you write aligned image bits? In some cases, it doesn't matter if the source bits are aligned, but if you're looking for speed in rendering your source, bits should be aligned. To write aligned image bits, the destination address must be 4-byte aligned. It looks like this:

```
destPtr = destPtr & 0xFFFFFFFC;
```

With these concepts in mind, you're almost ready to try your hand at setting Mode X using DirectDraw. There is one more small hurdle to jump, and you'll be ready to start playing with some DirectDraw surfaces.



To Blt() to Video Memory or Not to Blt() to Video Memory

With all these wonderful video cards hitting the market, such as the ATi and S3, I have become a lazy programmer. DirectDraw interfaces and video blitters do all the work for me. However, one afternoon while I was at work, I tried my Mode X code on a new Pentium system with an onboard ATi Mach 64 chip set. My code bombed miserably, and I couldn't figure out why my code worked

wonderfully at home on a Dell 486/66 with an ATi Mach 64 card. The next day I brought my Direct X CD-ROMs with me to work and installed the SDKs on the Pentium system to replicate my 486 development system at home. The code still bombed miserably. Just as I was about to toss the Pentium system out into the parking lot, I determined the only real difference between my system at home and the Pentium was the amount of video memory each system had. My ATi Mach 64 card in my Dell has 2M of VRAM, and the Pentium system onboard video only had 1M of VRAM. Surely, this suitable discrepancy couldn't be the root of the problem. Well, it was.

I was about to go insane with the different combinations of memory schemes I tried to get my Direct X Mode X to work on with only 1M of VRAM. Returning to the net for answers I managed to hook up with a developer at Raven Software, Rick Johnson, who provided some good insight on different things to try. A very important piece of advice from Rick was to go to the card vendor's Web site and get the new Windows 95 drivers that had just come out in the last few months. These had bug fixes on them. The pieces of the puzzle came together when Fred Hummel, a contract programmer for id Software working on *DOOM95*, told me a surefire way to initialize any DirectDraw surface. You start by trying to allocate a triple buffered configuration in video memory, and you keep backing off and trying lesser configurations, such as allocating surfaces in system memory, until one configuration finally works. I'll demonstrate the Fred Hummel hybrid *Foxbear* flip surface allocation scheme in a moment. The key lesson learned here is that DirectDraw is a video memory manager. Just as you check your malloc() call to ensure the memory was allocated, you must also clearly examine the HRESULT value returned when you request a DirectDrawSurface to be created. Many developers I talked to fell into the same pit because, as DOS programmers, we're used to checking pointers and not return result values.



The Mode X Renaissance

The common advice for using Mode X, from a Windows developer to a DOS programmer, is to just develop games that use 640 x 480 resolution. What the average Windows database developer doesn't understand is that the single most expensive component of game development isn't spent on the developer; it's spent on the art. Ken Nicholson, the Director of Game Technology at ATi, has preached on more than one occasion that it takes longer to port art and touch it up to a new screen resolution than it does to port the actual game code. He mentioned that the *SuperBubsy* port to Windows 95 took around a year while the art touch-ups required about 18 months of work.

A lot of game art has been around for five years or longer. I believe Rick Johnson at Raven Software mentioned he is porting one of his old games to Windows 95, which has artwork originally created on an Amiga back in 1991.

Because there is an investment in art in Mode X, it will be a popular game display mode for quite some time.



Initializing Mode X from DirectDraw

What I'm about to present is a slightly more complicated approach to initializing a DirectDraw object. It's more complicated in several ways. For one, I tried to replicate one of Mode X's bugs by running in a window rather than a whole screen so I could see error handlers write in a window. This effort left me with some code that allows my DirectDraw Mode X examples to run in a 320 x 240 window. I wanted it for debugging only, but it's still a neat effect so I left it in the code. When I initialize my DirectDraw object, I assume I have already initialized a *Spells of Fury* data structure I call the game console descriptor. The code for this is shown in Listing 12-2.

LISTING 12-2 *Spells of Fury* game console-style descriptor

```
// enumerate game console types
enum { GC_NONE, GC_FULLSCREEN, GC_WINDOW };
// game console capabilities desriptor
typedef struct {
  int  width;          // game console width
  int  height;         // game console height
  int  bpp;            // game console bits per pixel
  int  gcType;         // game console type FULLSCREEN | WINDOW
  int  nBuffers;       // backbuffer count
  BOOL bUseVideoMem;   // video or system memory
  BOOL bHardware;      // has hardware accelerators?
  BOOL bColorKey;      // has color key caps?
  BOOL bSrcColorKey;   // can use source color keys
}GCCapsDESC;
```

The first four fields are of interest when initializing a DirectDraw object. The first three fields are the *width*, *height*, and *bpp* of the game console you wish to create. The fourth field is the game console type, *gctype*. These values can be GC_FULLSCREEN, a full-screen 320 x 240 Mode X game console, or GC_WINDOW for a 320 x 240 Mode X windowed game console.

With these values initialized in the InitGameApp() function, called by WinMain(),you can create the desired DirectDraw object. The code for initializing a 320 x 240, with a 8 bpp depth looks like this:

```
gcdesc.width = 320;
gcdesc.height = 240;
gcdesc.bpp = 8;
gcdesc.gcType = GC_FULLSCREEN;
```

To initialize the DirectDraw object,you don't need these values. However, the *gcType* field is required to determine how to set the cooperative level of the

game application. You proceed by creating the DirectDraw object in the usual manner:

```
if (lpDD == NULL)
{
  if ((ddrval = DirectDrawCreate(NULL, &lpDD, NULL) != DD_OK) )
                                  //(LPGUID)DDCREATE_EMULATIONONLY
  {
    MessageBox(hwnd,"DirectDrawCreate Failed", "DXMODEX",MB_OK);
    return FALSE;
  }
}
```

Nothing spectacular in that line of code; it was seen before in Chapter 3. However, I actually do have a comment in this code for DDCREATE_EMULATIONONLY. This is a trick I picked up from Eric Engstrom, one of the architects of DirectDraw. If you recall from the *Foxbear* and *Roids* examples on the GDK and DirectX SDK CD-ROMs, this flag appeared in the code and it appeared as if there were a magic way for DirectDraw to enter an emulation only state. What is happening in *Foxbear* and *Roids* is that these demo applications can be forced into an emulation mode by executing the application from a DOS console command line using "-b" or some other parameter. When your code is misbehaving, just copy the DDCREATE_EMULATIONONLY and paste it over the first NULL parameter in DirectDrawCreate(). This will force your application to completely emulate DirectDraw operations and ignore hardware blitters and accelerators. It's a neat option for code debugging.



Cooperative Level and Display Mode for a Game Console Window

Now look at what's involved with setting the cooperative level for a windowed DirectDraw application. In my code to initialize DirectDraw, I have a switch statement that examines the game console type. If it is a windowed game console, *gcType* is set to GC_WINDOW. The code used is shown in Listing 12-3.

LISTING 12-3 Cooperative level and display mode for *gcType* = GC_WINDOW

```
case GC_WINDOW:
  if (gcSetCooperativeLevel(hwnd) != TRUE)
    return ModeXError("Window SetCooperativeLevel Failed");
  if (gcEmulateDisplay(hwnd) != TRUE)
    return ModeXError("Window SetDisplayMode Failed");
  break;
```

The code to set the window's cooperative level and display mode is broken up into two functions, gcSetCooperativeLevel() and gcEmulateDisplay(). The gc prefix identifies the function to a game console (gc). I use this in my code to differentiate game console routines from game console window routines. The function used to set the cooperative level for a game console window is shown in Listing 12-4.

LISTING 12-4 SetCooperativeLevel() for GC_WINDOW

```
BOOL gcSetCooperativeLevel( HWND hwnd )
{
  HRESULT ddrval;
  DWORD   dwCoopFlags = DDSCL_NORMAL;

  if (lpDD == NULL)
    return ModeXError("No DirectDraw Object");

  ddrval = lpDD->SetCooperativeLevel(hwnd, dwCoopFlags);

  if (ddrval != DD_OK)
    return ModeXError("Window SetCooperativeLevel Failed");

  gcdList[nModes].width = 320;
  gcdList[nModes].height = 200;
  if (nModes < MAX_MODE_ENTRIES)
    nModes++;  // number of available modes

  gcdList[nModes].width = 320;
  gcdList[nModes].height = 240;
  if (nModes < MAX_MODE_ENTRIES)
    nModes++;  // number of available modes

  return TRUE;
}
```

In this function, I made a simple list of display modes I wanted to emulate. If your application is going to toggle back and forth between modes and window emulation modes, as *DOOM 95* does, be sure this list is identical to your enumerated mode list that can be retrieved from DirectDraw (that is, use the same list if possible).

To emulate a display mode in a window, you have to put back into place all the ugly things you removed from the application in Chapters 1 through 3. You will need to place a title bar back on the window, change its attributes from WS_POPUP to WS_OVERLAPPED, and so on. Essentially, you're building back up a Windows application. Remember, the generic Windows 95 game application is 640 x 480. If you run a 320 x 240 display mode, you'll take up the entire screen. However, when you emulate a display mode in a window, 320 x 240 pixels is only one-fourth of the screen. You need to resize the application on the fly. Listing 12-5 shows the code used to achieve this.

LISTING 12-5 Emulating a display mode in a window

```
BOOL gcEmulateDisplay(HWND hwnd)
{
  HDC   hdc;
  RECT  gcRect;
  RECT  rc;
  RECT  workArea;

  // get the bit depth of the display
  hdc = GetDC(NULL);
  int deviceBPP = GetDeviceCaps(hdc,PLANES)
                    * GetDeviceCaps(hdc,BITSPIXEL);
  ReleaseDC(NULL, hdc);

  // check window style; is it in WS_POPUP (FULLSCREEN) mode?
  DWORD dwStyle = GetWindowStyle(hwnd);
  dwStyle &= ~WS_POPUP;  // change it from WS_POPUP
  dwStyle |= WS_OVERLAPPED | WS_CAPTION
                           | WS_THICKFRAME | WS_MINIMIZEBOX;
  SetWindowLong(hwnd, GWL_STYLE, dwStyle); // change window attribs

  SetRect(&gcRect,0,0,gcdesc.width,gcdesc.height);
  AdjustWindowRectEx(&gcRect, GetWindowStyle(hwnd),
                      GetMenu(hwnd) != NULL,
                      GetWindowExStyle(hwnd));
  SetWindowPos(hwnd,NULL,0,0,
                gcRect.right-gcRect.left,
                     gcRect.bottom - gcRect.top,
                     SWP_NOMOVE | SWP_NOZORDER | SWP_NOACTIVATE );
  SetWindowPos(hwnd, HWND_TOPMOST,0,0,0,0,
                SWP_NOSIZE | SWP_NOMOVE | SWP_NOACTIVATE );


  SystemParametersInfo(SPI_GETWORKAREA, 0, &workArea,0);
  GetWindowRect(hwnd, &rc);
  if (rc.left < workArea.left)
    rc.left = workArea.left;
  if (rc.top < workArea.top)
      rc.top = workArea.top;
  SetWindowPos(hwnd, NULL, rc.left, rc.top, 0,0,
         SWP_NOSIZE | SWP_NOZORDER | SWP_NOACTIVATE );

  return TRUE;
}
```



Mode X Full-Screen Cooperative Level and Display Settings

Having your virtual world run in a window is a nice feature. However, I take my gaming seriously and I don't want to see my e-mail icon flashing when I'm

blasting mutants. So my mode of preference is full-screen. To set the game application's cooperative level for a full screen using Mode X, the code looks very similar to what was used back in Chapter 3. Listing 12-6 shows how to set the cooperative level for a full-screen Mode X application using the flag DDSCL_ALLOWMODEX.

LISTING 12-6 SetCooperativeLevel() for full-screen Mode X

```
BOOL gcSetCooperativeLevel(HWND hwnd)
{
  DWORD dwFlags = DDSCL_NORMAL;

  if ((lpDD == NULL) || (hwnd == NULL) )
    return FALSE;

  // set the cooperative level for game console
  dwFlags = DDSCL_ALLOWMODEX | DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN;
  HRESULT ddrval = lpDD->SetCooperativeLevel(hwnd,dwFlags);
  lpDD->EnumDisplayModes(0,NULL,0,EnumModes);

  // set the cooperative level for windowed game console
  ddrval = lpDD->SetCooperativeLevel(hwnd,dwFlags);

  if (ddrval != DD_OK)
    return FALSE;
  return TRUE;
}
```

In Listing 12-6 you'll notice that I called the DirectDraw method EnumDisplayModes(). This isn't required when you set the cooperative level. So why is this code here in the first place? When I first wrote this code, I was experiencing problems on my AST 486 system that was resurrected from its duties as a doorstop. The AST has onboard VGA, an ATi Mach 32 chip with 1M of memory. My Mode X code wouldn't work in full-screen mode on the AST; the video memory seemed to behave differently on several machines. Before I arrived at this conclusion, it was 2:00 A.M. and my mind wasn't functioning properly. So I decided to use DirectDraw to query the interface and tell me which video modes were available through DirectDraw. The method EnumDisplayModes() did this using a CALLBACK function supplied by the programmer. Listing 12-7 shows the CALLBACK function I used to build a table of DirectDraw supported video modes on the current device.

LISTING 12-7 Querying your hardware for display modes

```
#define MAX_MODE_ENTRIES 25  // arbitrary guess of available modes
// game console mode descriptor
typedef struct {
  int width;       // video mode height
```

continued on next page

continued from previous page

```
  int  height;     // video mode width
  int bpp;         // video mode bits per pixel
} GCModeDESC;
GCModeDESC  gcdList[MAX_MODE_ENTRIES];  // list of avail video modes

HRESULT CALLBACK EnumModes(LPDDSURFACEDESC lpddsd, LPVOID context)
{

  gcdList[nModes].width = lpddsd->dwWidth;
  gcdList[nModes].height = lpddsd->dwHeight;
  gcdList[nModes].bpp = lpddsd->ddpfPixelFormat.dwRGBBitCount;
  if (nModes < MAX_MODE_ENTRIES)
    nModes++;  // number of available modes
  return DDENUMRET_OK;
}
```

In a later function, I wrote this list out to disk for debugging. I chose to leave this debug code in *Spells of Fury* because I found other software developers were doing similar things for their debugging purposes. You'll find a MODES.LST file when you run the demo that tells you which video modes DirectDraw supports on your device. This information told me that DirectDraw did support Mode X on my ATi Mach 32 video, and that my problems existed elsewhere in my code. This problem will be reexamined when I discuss creating flipping surfaces for Mode X. Listing 12-8 shows the function InitDXModex(), which I use to initialize a Mode X game console window or full-screen display.

LISTING 12-8 Initializing DirectDraw for Mode X

```
BOOL InitDXModeX(HWND hwnd)
{
  HRESULT ddrval;  // DirectDraw Return Value

  gcType = gcdesc.gcType;
  // first try to create a DirectDraw object
  if (lpDD == NULL)
  {
    if ((ddrval = DirectDrawCreate(NULL, &lpDD, NULL) != DD_OK) )
    //(LPGUID)DDCREATE_EMULATIONONLY
    {
         MessageBox(hwnd,"DirectDrawCreate Failed", "DXMODEX",MB_OK);
      return FALSE;
    }
  }

  switch(gcType)
  {
    case GC_NONE:
        break;

      case GC_FULLSCREEN:
```

```
      // SetCooperativeLevel
      if (gcSetCooperativeLevel(hwnd) != TRUE)
          return ModeXError("SetCooperativeLevel Failed");

      // SetDisplayMode
      if (gcSetMode(gcdesc.width,gcdesc.height,gcdesc.bpp) != TRUE)
        return ModeXError("gcSetMode Failed");
        break;

       case GC_WINDOW:
        if (gcSetCooperativeLevel(hwnd) != TRUE)
          return ModeXError("Window SetCooperativeLevel Failed");
        if (gcEmulateDisplay(hwnd) != TRUE)
          return ModeXError("Window SetDisplayMode Failed");
        break;

       default:
        break;
  }
  // everything is fine so far
  return TRUE;
}
```

The function InitDXModex() allows initialization of a DirectDraw object and a display or window for Mode X. The topic of the next discussion is creating the DirectDraw flipping surfaces for Mode X.



DirectDraw Flipping Surfaces for Mode X

Writing graphics routines using DirectDraw is fairly simple and straightforward. I made the assumption that since a 640 x 480, 8 bpp, display mode was so easy to configure, a 320 x 240, 8 bpp, display should be a no-brainer. Boy, was I ever wrong! When I first started coding the example program *3DTomb*, I knew I wanted it to be in a 320 x 200 or a 320 x 240 Mode X. The graphics card I used was an S3 developer's card. The card had 2M of VRAM, which allowed me to make silly mistakes. When I was cutting and pasting code from DDTOOLS, I didn't stop to think that a 320 x 240 triple-buffered display would fit nicely in this card's VRAM. When I tested the code on my AST, the demo didn't work at all. The problem was that the AST had an onboard VGA ATi chip set with only 1M of VRAM. This device couldn't handle a triple-buffered Mode X flip surface in VRAM.

My first assumption about the problem was that for some strange reason the video drivers on this AST system didn't support DirectDraw Mode X. This is when I incorporated the DirectDraw method enumDisplayModes(). I was hoping that a 320 x 240, 8 bpp display mode wouldn't be in the output list. Well, to my dismay, it was. This left only one other possibility as to my problem: the way I managed my DirectDraw flipping surfaces was completely wrong.

My first stab at the problem involved calling the DirectDrawSurface method GetCaps() once I created the primary surface. From there I examined the memory size that was reported to DirectDraw. Based on the value in the DIRECTDRAWSURFACEDESC field, I would then create my back buffers from system or video memory depending on this constraint.

This clever scheme failed miserably as well. By now I was becoming pretty frustrated and I was purposely plowing my car into my neighbors' garbage cans on my drive into work. I sent out an e-mail to a few developers and received a response back from Fred Hummel, who had already dealt with this problem in his work on *DOOM95*. Fred's information is proprietary, so he tends to talk in tongues and riddles. But he gave me two significant pieces to the puzzle. For one, he pointed out that I should look at how the *Foxbear* demo allocated its surfaces. Fred mentioned he used this as a model for *DOOM95*. Second, he said his scheme for allocating surfaces was to try for the golden surface configuration of triple-buffered video memory and then keep falling back until one scheme works. If you can't even get a single back buffer in system memory, then your system is completely messed up. This technique worked for me. So, in honorarium for Fred, I call this the Fred Hummel hybrid *Foxbear* surface allocation scheme. I can't pay you Fred, but I can put your name in print.



Creating DirectDraw Mode X Surfaces

When creating the primary flipping surface, there are two cases to handle. First, the primary surface is in a window and you must use a DirectDrawClipper object. Second, you are using a full-screen mode, and you must try to set up the flipping surfaces in video or system memory. In this section, you'll look at how both of these cases are handled by the example programs for Chapter 12.



Primary Surface for a Window

When using DirectDraw in a window, you won't be able to page flip. Working with DirectDraw in a window is a lot like working with mode 13h. Mode 13h, as it's presented by BIOS, isn't a page-flipping display mode. Programmers typically create an offscreen system memory buffer and blit the contents to the displayable surface at location 0xA000. This is the same thing the window surface will do. The primary surface is the displayable surface. Create the

lpDDSBack buffer using the DDTOOLS routine GetNewSurface(). Listing 12-9 shows how I set up the Primary and Back buffer for use in a window.

LISTING 12-9 Creating surfaces for a window

```
HRESULT         ddrval;
DDSURFACEDESC   ddsd;
DDSCAPS         ddscaps;
DDCAPS          ddcaps;
DWORD           dwFlags;


memset( &ddsd, 0, sizeof(ddsd) );
ddsd.dwSize = sizeof( ddsd );
memset( &ddcaps, 0, sizeof(ddcaps) );
ddcaps.dwSize = sizeof( ddcaps );
nBackBuffers = 0; //GCS_WINDOW;
ddsd.dwFlags = DDSD_CAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
ddrval = lpDD->CreateSurface(&ddsd, &lpDDSPrimary, NULL);
if (ddrval != DD_OK)
  return ModeXError("Create Window lpDDSPrimary Failed");
// create the back surface
lpDDSBack = GetNewSurface( gcdesc.width, gcdesc.height );
if (lpDDSBack == NULL)
  return ModeXError("Can't Create Window lpDDSBack Failed");
```

Once you have the DirectDrawSurfaces, create a DirectDrawClipper object. The DirectDrawClipper object is similar to a DirectDraw object, only this object owns a display window. Why do you need a clipper object? Well, say I'm playing *DOOM95* at work and waiting for the e-mail icon in the top-right corner of my screen to start flashing. I may also have an X-terminal window open with a connection that I am using to compile some code on a UNIX workstation. This is a case where I don't want full-screen graphics and page flipping. I only want a window open to play a game and kill some time until a process finishes. The DirectDrawClipper object will allow my game to use DirectDraw in a nonexclusive mode, as shown in Figure 12-3.

With the Primary and Back surfaces created, now initialize the DirectDrawClipper object. The code for this is shown in Listing 12-10. You must initialize a clipper object, and then instruct the object which window to send DirectDraw information to.

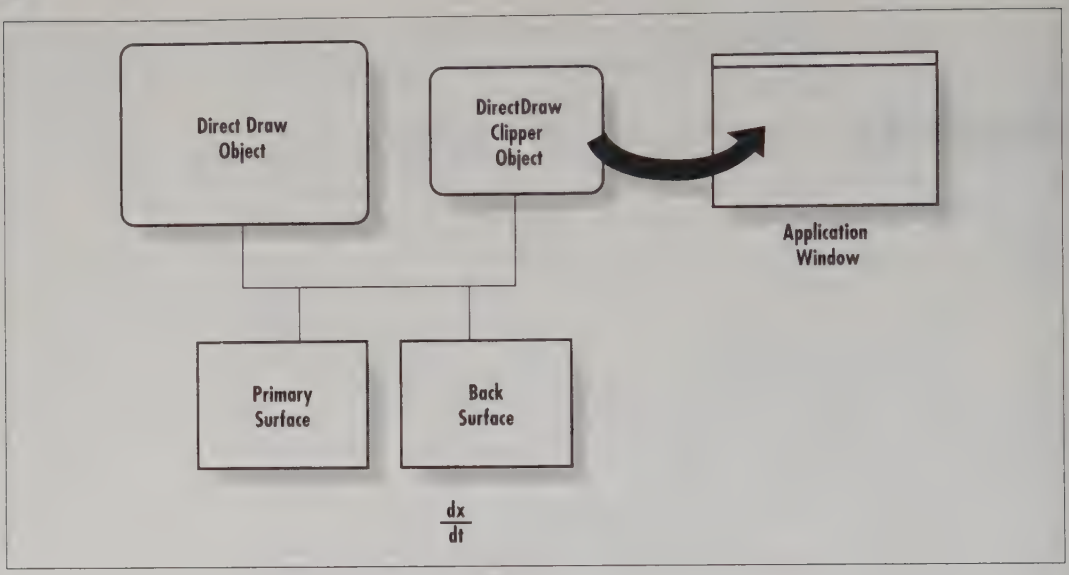


FIGURE 12-3 A DirectDrawClipper object

LISTING 12-10 Initializing the DirectDrawClipper object

```
// create the DirectDrawClipper object
ddrval = lpDD->CreateClipper(0, &lpDDClipper, NULL);
if (ddrval != DD_OK)
  return ModeXError("CreateClipper Failed");

// set the clipper to reference window that will receive clipper data
ddrval = lpDDClipper->SetHWnd(0,hwnd);
if (ddrval != DD_OK)
  return ModeXError("Set Clipper HWND Failed");

// set the clipper to the window
ddrval = lpDDSPrimary->SetClipper(lpDDClipper);
if (ddrval != DD_OK)
  return ModeXError("SetClipper()  Failed");
EraseSurface(lpDDSPrimary);
EraseSurface(lpDDSBack);
gcdesc.bUseVideoMem = FALSE;  // uses system memory
```

The code shows that the DirectDrawClipper was initialized in three steps. First, you initialize the clipper object to use for the window. Second, you pass the handle of the window you want to update to the DirectDrawClipper object. And last, set the primary surface to the clipper object. That's all that's needed to get the window up and running with DirectDraw.

Creating Flipping Surfaces for Mode X

The methodology presented here for creating Mode X flipping surfaces is actually useful for creating surfaces in any mode. I will present it here because this is what I used for my Mode X examples here in Chapter 12. In Chapter 3, a primary surface was created with one back surface for flipping surfaces. Two 640 x 480 surfaces with a depth of 8 bpp will sit rather nicely in any VGA card with 1M or better. This technique is demonstrated in almost all of the DirectDraw examples on the Direct X CD-ROMs. Now, what's not so obvious is how to handle double and triple buffers using a display setting such as Mode X. As I mentioned, the nice newfangled 2M developer cards handle this rather nicely. However, it is common for the consumer to have a motherboard with an onboard VGA chip set such as the S3 or an ATi with only 1M of memory. Intel and IBM are now hopping onto the multimedia bandwagon and providing onboard hardware with 2M VRAM. But the problem still exists, such as with my AST 486, which has only 1M of video memory.

To address this issue, I pretty much beat my head against a wall trying to figure out a clever procedure to determine when I should allocate a surface from video memory or when I should allocate from system memory. In the end I found that Fred's method works the best, and I am going to present it here. This procedure involves setting the most ideal flipping surface configuration, triple-buffered in video memory. If you're lucky, you're sitting on an S3 or an ATi card which can handle this configuration. If it fails, try the next best configuration, a double-buffered scenario in video memory. It just might work. In the event this fails, set up the back buffer in system memory. If this fails, the system really has some problems. Figure 12-4 shows a simple scenario of what's happening in this scheme of allocation.

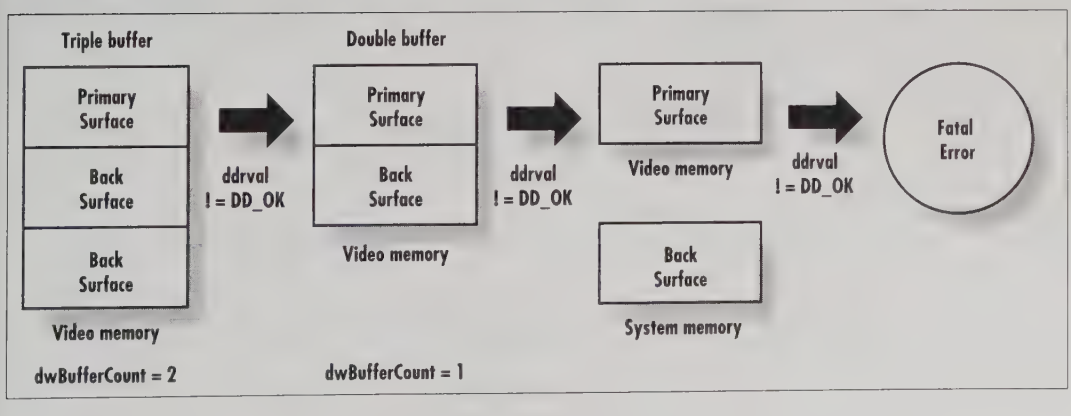


FIGURE 12-4 Allocating flipping surfaces by best fit

The source in Listing 12-11 shows how to use the best fit scheme to allocate the flipping surfaces. What's clever about this scheme is that you don't have to know ahead of time which memory type the surface is going to use. It's brute force, but it works well.

LISTING 12-11 Implementing the best fit allocation scheme

```
if (gcType == GC_FULLSCREEN)
{
  fprintf(fdebug,"GC_FULLSCREEN\n");
  // Create the primary surface with 2 back buffers
  ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
  dwFlags =DDSCAPS_PRIMARYSURFACE
           | DDSCAPS_FLIP | DDSCAPS_COMPLEX | DDSCAPS_VIDEOMEMORY;
  ddsd.ddsCaps.dwCaps = dwFlags;
  ddsd.dwBackBufferCount = 2;
  ddrval = lpDD->CreateSurface( &ddsd, &lpDDSPrimary, NULL );
  if( ddrval != DD_OK ) // triple buffer create failed
  {
    fprintf(fdebug,"triple buffer create failed\n");
    // if triple buffer failed, can we create a double buffer?
    ddsd.dwBackBufferCount = 1;
    ddrval = lpDD->CreateSurface( &ddsd, &lpDDSPrimary, NULL );
    if (ddrval != DD_OK) // double buffer create failed
    {
      fprintf(fdebug,"double VIDEO buffer create failed\n");
      // try double buffer in system memory
      ddsd.ddsCaps.dwCaps &= ~DDSCAPS_VIDEOMEMORY;
      ddrval = lpDD->CreateSurface( &ddsd, &lpDDSPrimary, NULL );
      if (ddrval != DD_OK)
      {
        fprintf(fdebug,"FATAL double system buffer create failed\n");
        return ModeXError("Can't Allocate FLIP Surf in any Memory");
      }
    } // double buffer failed
  }  // triple buffer failed

  fprintf(fdebug,"ddsd.dwBackBufferCount %d\n",
                        ddsd.dwBackBufferCount);

  // get the backbuffer
  ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
  ddrval = lpDDSPrimary->GetAttachedSurface( &ddscaps, &lpDDSBack);
  if (ddrval != DD_OK)
  {
    fprintf(fdebug,"FATAL: BackBuffer not found %d\n",ddrval);
    return ModeXError("BackBuffer not found");
  }
  else
    fprintf(fdebug,"BackBuffer found %d\n",ddrval);
  // what memory is the back buffer?
```

```
  gcdesc.bUseVideoMem = FALSE;

  DDSCAPS ddsBack;
  ddrval = lpDDSBack->GetCaps(&ddsBack);
  fprintf(fdebug,"lpDDSBack->GetCaps() %d\n",ddrval);
  if (ddrval == DD_OK && ddsBack.dwCaps & DDSCAPS_VIDEOMEMORY)
  {
    fprintf(fdebug,"dwCaps video memory %d\n",ddrval);
    gcdesc.bUseVideoMem = TRUE; // backbuffer is in video memory
  }
  if (ddrval == DD_OK && ddsBack.dwCaps & DDSCAPS_SYSTEMMEMORY)
  {
    fprintf(fdebug,"dwCaps system memory %d\n",ddrval);
    gcdesc.bUseVideoMem = TRUE; // backbuffer is in video memory
  }
} // gcType = GC_FULLSCREEN
```

You'll notice that I have another debug routine in this body of code to write what surfaces were allocated. This information is written out to a file called FLIP.LST. I decided to leave this in the *Spells of Fury* code since it helped me debug what was happening in different video configurations.



Updating the Game Console

The method of rendering to the game console has become slightly more complicated than the DDTOOLS routines. Updating the game console now has two cases to contend with. If it's a full-screen game console you will surface flip. If you are in a window, update the primary surface with the offscreen back surface using a blit operation. Listing 12-12 shows the new code for updating the screen.

LISTING 12-12 Updating the game console

```
BOOL FlipSurface( void )
{
  if (gcType = GC_FULLSCREEN)
  {
    HRESULT ddrval = lpDDSPrimary->Flip(NULL, DDFLIP_WAIT);

    if (ddrval != DD_OK)
      return FALSE;
  }
  else
  {
    if (UpdatePrimarySurface() == FALSE)
      return FALSE;
  }
  return TRUE;
}
```

continued on next page

continued from previous page

```
//////////////////////////////////////
// UpdatePrimarySurface (Blt)
//////////////////////////////////////
BOOL UpdatePrimarySurface( void )
{
  RECT srcRect;

  SetRect(&srcRect,0,0,gcdesc.width,gcdesc.height);

  HRESULT ddrval = lpDDSPrimary->Blt(&srcRect,
                lpDDSBack, NULL, DDBLT_WAIT, NULL);
  if (ddrval != DD_OK)
    return FALSE;
  return TRUE;
}
```

These graphics routines handle the virtual problems the virtual reality viewing hardware will encounter. This code alone was the single biggest headache I encountered for preparing these examples. But put this behind you now and we'll move on to some cool things: polygons!



Filled Polygons

In the 1400s, there lived an artist named Paolo Uccello, who rarely received important commissions for his work. Other artists of the Renaissance were receiving recognition for frescos, altar pieces, and paintings. Uccello painted little, though when he did, his work was exquisite. Art critics comment that Uccello was obsessed with perspective geometry and wasted too much time on his studies. He was deemed nothing more than a doodler. As an armchair art critic, I say Uccello was far more than a doodler. One of Uccello's pen and ink studies on perspective is a chalice constructed entirely of polygons. His studies in perspective and composition could very well make him a visionary of virtual reality, just as Leonardo da Vinci is regarded as a visionary of flight. The work of Uccello could very easily be placed on the cover of a Foley and Van Dam computer graphics textbook and be mistakenly regarded as being computer-generated. People take for granted that polygons are a modern day interpretation of space and volumes. But people long ago used this tool in pen and ink, yet they never mentioned the words words *floating* and *fixed point*.

So why a discussion on filled polygons when the topic is virtual reality? Just as the painters of the Renaissance were interested in lines, polygons, perspective, and composition, so too is the present day computer graphics programmer. The notions and inklings to perceive 3D space and convey it in a real form are just the same today as they were centuries ago.



3D Composing on a 2D Surface

In college, one of my favorite classes was drawing. For me, it was a therapeutic experience. What other class would give you a grade for sitting outside on a beautiful spring day to sketch trees and buildings. As relaxing as the class may have been, I did walk away with a very important lesson from my art instructor. Whenever I drew a building or a landscape, I would start in the top-left corner and fill in all the detail such as shading and lighting and pan across to the right and finish the drawing. Perhaps, as a graphics programmer, my brain was stuck in a column-drawing mode. My art instructor told me that I obviously had a grasp of perspective, but I should spend more time in composition and breaking up of the 2D space.

When I started programming in 3D, I wanted to pull in the knowledge of the art community as well as what I was taught in linear algebra. After all, artists had been studying perspective centuries longer than programmers have. My first objective in 3D design was to develop tools to allow me to study composition. This is the same thought process I believe Uccello used when he rendered his pen and ink polygon chalice. He wasn't interested in how light would reflect. He was interested foremost in the composition and breaking up the space on paper.

In my analysis, this meant I needed a tool that was flexible and could be up and running quickly. For this reason I chose not to use texture maps. Filled polygons would allow me to check my 3D space and perspective. If the angles were right and the viewer frustum was okay, I would go back and add the texture mapping later.



Convex Polygon Scan Conversion

The design requirements for my filled polygon rendering were speed and the ability to write scanlines rather than draw columns. Writing scanlines would take advantage of the DirectDrawSurface flat memory architecture and possibly minimize the number of surface locks I would have to perform.

These requirements seemed like a good idea at first, but the more I thought about how Foley and Van Dam described polygon filling, the more I thought my design requirements were unrealistic. In *Introduction to Computer Graphics*, by Foley and Van Dam, a polygon is filled by using directional vectors to walk the edges of the polygon. If the directional vector is negative, the algorithm is walking down an edge, and if the directional vector is positive, it's walking up an edge, as shown in Figure 12-5. The idea is that a directional vector winds its way around the polygon and fills in an Active Edge Table.

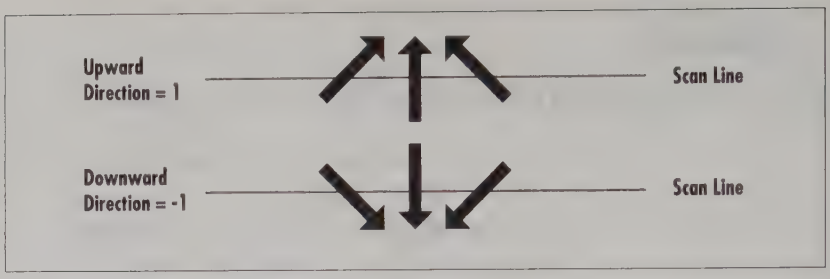


FIGURE 12-5
Filling polygons with winding vectors

The more I read on about this algorithm, the more my head began to hurt. I wrote simple programs on my Sun workstation to implement Foley's ideas. Then one afternoon at lunch it dawned on me that I could do something clever with the tools I had already written for Chapter 6 using the Bresenham line-drawing algorithm. I already knew I was going to use a 320 x 240 resolution screen mode, and I knew the maximum number of scanlines I was going to write to the offscreen DirectDraw surface. So, why couldn't I create a logical array of scanlines and use the Bresenham code to write edges into the logical array, shown in Figure 12-6? This meant I could use simple filtering rather than complex clipping to load values in the array. The first filter I would use is *y* boundary checking. I did this because the *y* coordinate is the index into the logical array. The fields in the array are the minimum *x* value and the maximum *x* value of the scanline in the polygon. If the *x* value is greater than 319 or less than 0, it is clipped.

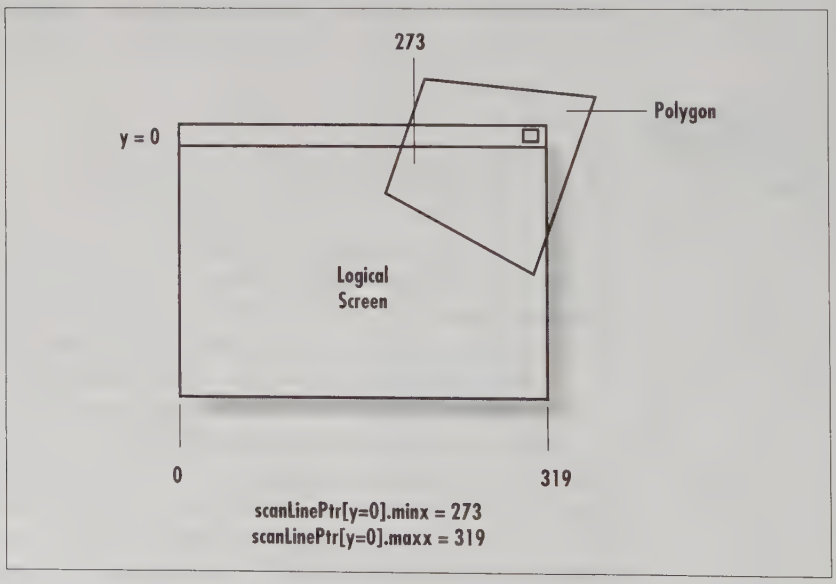


FIGURE 12-6
Scanline conversion in a logical array

The majority of work in this scheme would be carried out using a modified version of the Bresenham line-drawing algorithm presented earlier. The original code was called BresenhamLine(). I renamed this one to BresenhamEdge() since its function will be to draw the edges of the convex polygon. The code to BresenhamLine() is identical to BresenhamEdge(). I changed the name in file RENDER.CPP so I would know that the overall functionality has been modified. Listing 12-13 shows the code for the function BresenhamEdge().

LISTING 12-13 The Bresenham line-drawing algorithm

```
void BresenhamEdge(int x1, int y1, int x2, int y2, BYTE color)
{
    int dx, dy;
    int step;       // direction to proceed

    // if y1 > y2 swap coordinates
    if (y1 > y2)
    {
      swapTermsInt(&y1, &y2);
         swapTermsInt(&x1, &x2);
    }

    // calculate deltas
    dx = x2 - x1;
    dy = y2 - y1;

    if (dx > 0)
    {
      step = 1; // proceed in the positive direction
    }
    else
    {
      dx = -dx;
      step = -1;
    }

    if (dx > dy)
        ApproxY( x1, y1, dx, dy, step, color );
    else
        ApproxX( x1, y1, dx, dy, step, color );

}
```

The difference in this code and the Chapter 6 flavor is that ApproxX() and ApproxY() have been modified to write to the logical array and not to the back buffer DirectDraw surface. Listing 12-14 shows my *x* and *y* approximation routines. This code is also straight from the pages of Chapter 6.

LISTING 12-14 Approximating the next *x* and *y* coordinates

```
////////////////////////////////////////
// Approximate Next Y Pixel
////////////////////////////////////////
void ApproxY( int xp, int yp, int dx, int dy, int dirVector, BYTE color )
{
  int k1, k2, errorTerm;

  k1 = dy << 2;
  k2 = k1 - (int)(dx << 2);
  errorTerm = k1 - (int) dx;
  setScanLineEntry( xp, yp, color );

  while (dx--)
  {
     if (errorTerm >= 0)
        {
          yp++;
          errorTerm += k2;
        }
        else
          errorTerm += k1;
        xp += dirVector;
        setScanLineEntry( xp, yp, color );
  }
}

////////////////////////////////////////
// Approximate Next X Pixel
////////////////////////////////////////
void ApproxX( int xp, int yp, int dx, int dy, int dirVector, BYTE color )
{
  int k1, k2, errorTerm;

  k1 = dx << 2;
  k2 = k1 - (int)(dy << 2);
  errorTerm = k1 - (int) dy;
  setScanLineEntry( xp, yp, color );

  while (dy--)
  {
     if (errorTerm >= 0)
        {
          xp += dirVector;
          errorTerm += k2;
        }
        else
          errorTerm += k1;
        yp++;
        setScanLineEntry( xp, yp, color );
  }
}
```

The only difference in this code is that the values are written into the logical scanline array. Where, in Chapter 6, it had writeSurface() calls, it now has setScanLineEntry() calls. Listing 12-15 shows the code for setScanLineEntry().

LISTING 12-15 Setting the scanline entries in the array

```
struct ScanLineEntry
{
  int  xmin;       // first x value in scan line
  int  xmax;       // last x value in scan line
  int  surfaceRow; // the row to draw this scan line
  int  byteRun;    // draw n bytes to this scan line
  int  color;      // color to draw to this scan line
  BOOL bFlag;      // if scan line is inbounds (1) draw it
};

struct ScanLineEntry *scanLineListPtr;

void setScanLineEntry( int x, int y, int color )
{
  int ix = y;


  if (ix < 0)   // check for under flow
    return;     // bad index
  if (ix > 240)     // check for over flow
    return;     // bad index
  // set first point
  if (scanLineListPtr[ix].bFlag == FALSE)
  {
    scanLineListPtr[ix].xmin = x;
    scanLineListPtr[ix].xmax = x;
    scanLineListPtr[ix].bFlag = TRUE;
  }

  ////
  // check for new xmin and xmax
  ////

  // check for min x ( x < 0 )
  if (x < scanLineListPtr[ix].xmin)
    scanLineListPtr[ix].xmin = x;

  // check for new xmax value
  if (x > scanLineListPtr[ix].xmax)
    scanLineListPtr[ix].xmax = x;

    ////
  // if xmin in bounds is xmax?
  ////
  if ( (scanLineListPtr[ix].xmin >= 0) )
  {
```

continued on next page

continued from previous page

```
    if ( (scanLineListPtr[ix].xmin < VIEW_PORT_MAX_X)
         && (scanLineListPtr[ix].xmax >= VIEW_PORT_MAX_X) )
      {
       scanLineListPtr[ix].xmax = VIEW_PORT_MAX_X - 1;
      }
  }

  scanLineListPtr[ix].surfaceRow = y; // the screen row to write scan line
  scanLineListPtr[ix].color = color;  // set the color

}
```

The function of setScanLineEntry() is to act as a filter when you enter the edge values into the array. The *y* component of the edge is the index into the array. You are dealing with convex polygons only, so at most you'll have two points for each scanline. If the *y* value is negative or greater than 239, it won't be entered in the scanline list. This is the method for filtering the *y* values by scanline. If the *y* values are valid, proceed to clip the *x* values if necessary. As the *x* values come in, also check to see if the new *x* value is an *xmin* value or an *xmax* value. A swap takes place if necessary.



Drawing the Filled Convex Polygon

Now that there is a tool to generate the polygon, it's time to focus on drawing it to the offscreen DirectDraw surface. The function to call to draw a filled polygon is floodFillPolygon(). The source for this function is shown in Listing 12-16.

LISTING 12-16 Drawing a filled polygon

```
#define MAX_VERTEX_ENTRY   4     // 4 vertices describe a wall polygon
struct Vertex
{
  int x;  // x coordinate of vertex
  int y;  // y coordinate of vertex
};
struct Vertex  *vertexListPtr;   // list containing vertices of a polygon


BOOL floodFillPolygon(struct Vertex* lpv, int color)
{

  if ((numScanLines = getScanLineCount(lpv)) <= 0)
    return FALSE;  // no scan lines in polygon to fill
 memset(scanLineListPtr,0,sizeof(struct ScanLineEntry)*240);

  BresenhamEdge(lpv[0].x, lpv[0].y, lpv[1].x, lpv[1].y, color);
  BresenhamEdge(lpv[1].x, lpv[1].y, lpv[2].x, lpv[2].y, color);
  BresenhamEdge(lpv[2].x, lpv[2].y, lpv[3].x, lpv[3].y, color);
```

```
  BresenhamEdge(lpv[3].x, lpv[3].y, lpv[0].x, lpv[0].y, color);
  // draw the scan line list
  drawScanLineList();
  return TRUE;
}
```

The function works by calling BresenhamEdge() on all the vertices of the polygon. This will build the entire filled polygon in the scanline list. Once all the vertices have been written to the scanline list you can write the array to the Back DirectDraw surface. This function can be cleaned up so you loop through the vertices. This would allow drawing triangles as well as trapezoids. For now, the simplicity helps convey what the function is doing. The function drawScanLineList() is used to write the logical array to the Back DirectDraw surface, shown in Listing 12-17.

LISTING 12-17 Drawing the array to the back surface

```
void drawScanLineList( void )
{
  int ix;
  int x,xmax, y, cBytes;
  DDSURFACEDESC DDSDesc;
  LPSTR lpDstBits;
  HRESULT ddrval;

  DDSDesc.dwSize = sizeof(DDSDesc);
  // lock the surface for a write
  while(1)
  {
    ddrval = lpDDSOffScreen->Lock(NULL, &DDSDesc, 0, NULL);

    if(ddrval == DD_OK)
      break;
    if(ddrval != DDERR_WASSTILLDRAWING)
      return;
  }

  for (ix = ymin; ix < ymax; ix++)
  {
        x = scanLineListPtr[ix].xmin;
        xmax = scanLineListPtr[ix].xmax;
        if (xmax > VIEW_PORT_MAX_X)
        {
          if ( (x >= 0) && (x < VIEW_PORT_MAX_X) )
              x = VIEW_PORT_MAX_X - 1;
        }
        y = scanLineListPtr[ix].surfaceRow;
        scanLineListPtr[ix].byteRun = xmax - x;
        cBytes = xmax - x;

    lpDstBits = (LPSTR) DDSDesc.lpSurface + (y*SCREEN_PITCH);
```

continued on next page

continued from previous page

```
        // write to the surface
        if ( (scanLineListPtr[ix].byteRun > 0)
           && ((scanLineListPtr[ix].xmin +
                        scanLineListPtr[ix].byteRun) < 320))
        {
          memset(lpDstBits + x, scanLineListPtr[ix].color,
                                scanLineListPtr[ix].byteRun);
        }
    } // for ix
    // unlock the surface
    lpDDSOffScreen->Unlock(NULL);
}
```

As you may have noticed from the code in Listing 12-16, I am bounds checking once more to see if a scanline needs to be clipped. "Gosh, Mike, have you gone mad?" My answer is yes and no. When I wrote this portion of the code, I ran into the chicken and the egg dilemma. In the function setScanLineEntry(), I was checking to see if the *x* value coming in was an *xmin* or *xmax* value. There was no way in that function to determine if I had both my *xmin* and *xmax* values and if their filtering had stabilized. I had to wait until the polygon generation process completed. Of course, the next step, once the polygon was generated, was to draw it. So here in function drawScanLineList(), I do the final bounds checking I was unable to perform in setScanLineEntry().



Rendering Realities with Filled Polygons

Polygons are fun to play with, but I got bored real fast just rendering one simple polygon to the screen for my tests. I wanted to take a leap of programming faith and try building a world of polygons. I remembered a series of articles in *Doctor Dobb's Journal*, by Michael Abrash, on Binary Space Partitioning (BSP) trees. I believe they were the June, August, and October 1995 editions. Abrash clearly published a method to build a 3D world using BSP trees. The listings in the journal were long and incomplete due to space constraints. However, in the October 1995 edition, Abrash mentioned he threw the code for the BSP compiler out on the net. This was perfect, because I didn't want to go out of my way piecing together the compiler from excerpts of his code. So I downloaded his compiler and played around with it. As I got more confident, I decided to make a cool landscape for a 3D world. In my art history books, the floor plans to Renaissance cathedrals were laid out to scale. I thought what the heck, I'll do homage to Renaissance artists and put a cathedral in my 3D space. It took me about an hour to translate by hand on quadrille paper the Amiens cathedral in France over to the Abrash BSP compiler. The scale was a bit off, but the final result was pretty impressive. So, as you travel through the demo in DXMODEX and 3DTOMB, this is actually the floor plan to the Amiens cathedral. The walls in the 3D space seem weird. I was trying to do a study to get a colossal effect

as you went through the tomb. I didn't quite succeed as well as Camarack and Abrash did with *Quake*. But it was a cool study. I attempted to use one of the tombs from Egypt but I quickly exceeded the maximum number of nodes the Abrash BSP compiler was capable of handling.



Virtual I/O Stereoscopic Headset

Where would a virtual reality discussion be without a section on stereoscopic glasses? When I conducted my search for a virtual reality headset, I first and foremost wanted a device that was DirectInput compatible. I met with developers and engineering directors of stereoscopic headset manufacturers and several problems surfaced immediately in our discussions. For one, DirectInput provides no means for bidirectional communication between devices. Virtual reality headsets need to send floating point values back up through the interface. I spoke with some of the most visionary people in the industry and got to share a lot of interesting ideas. Microsoft is listening to these manufacturers and is currently drafting a DirectInput 2.0 interface to address these vendors' needs.

What are the problems with the current DirectInput interface? As was demonstrated to me at Forte, the developers of VFX1 headset, a virtual reality game requires input from the user's head movements, as well as a game controller interface. For instance, in a *MechWarrior 2* demonstration, the headset controlled the mech's point of view, while the control pad handled the weapons and the movement. This was a very intricate, yet usable interface. A lot of work is being done in this field, and with the introduction of DirectInput 2.0, you'll see a lot of high-tech toys, once only in science fiction films, enter our households.

For this demonstration, I selected the Virtual I/O *i-glasses* headset. At the time of this writing, the Virtual I/O API supported Win32 and Visual C++ 4.0. The Win32 library was a new feature. Developers at Virtual I/O and developers at Microsoft were working on a DirectInput interface. The development team at Forte was also working on a DirectInput driver, but the VFX1 required the bidirectional communication scheme not available in DirectInput 1.0. The Virtual I/O *i-glasses* had the same problems. However, theVirtual I/O device did support serial communication in the Win32 API and was suitable for my needs.



Mechanics of Virtual Reality

There are two industry buzzwords used today that have ambiguous definitions. The first term is virtual reality and the other is multimedia. What is virtual reality? Science fiction and Hollywood have various definitions of this. In the

1980s Disney hit *Tron*, a programmer was transported into a binary world inside a computer using a laser to digitize the reluctant traveler. Recent movies such as *Lawnmower Man* and *Virtuosity* used stereoscopic headsets for their characters to interact with the world inside the computer. The industry is catching up with Hollywood in the latter sense. Head-tracking systems, providing real-time input for stereoscopic viewing of 3D space, are entering the marketplace. Currently, virtual reality technology for the consumer consists of headsets with stereoscopic vision, a head tracker, and now even a hand tracker with controls.



Tracking the Movement of the Player's Head

Virtual reality headsets track movement by taking real-time readings of the player's head attitude in the earth's magnetic field. All measurements are made from the perspective of the headset user. Displacements are reported from the device using orientations of pitch, roll, and yaw, as shown in Figure 12-7. For those of you familiar with aerodynamics and astrodynamics, these are values used to determine object attitude and orientation in a 3D space. Pitch measures the movements in the up and down direction, or the *yz* plane. Roll measures the change of tilt from side to side, or the *xy* plane. Yaw is the horizontal change from side to side on the *xz* plane. These values use the right-hand rule for a coordinate system. If this terminology seems unfamiliar to you, consult *Introduction to Computer Graphics*, by Foley, or *Black Art of 3D Game Programming*, by LaMothe. In the right-handed system, the $+y$ axis is up, $+z$ is out of the page, and $+x$ is to the side.



Virtual Frame of Reference

The virtual world is a logical cube 32,000 x 32,000 units wide on the *xz* plane with a maximum height of 32,000 units on the *y* axis. Snapping our rectangular coordinate system into this bounding cube, we give the *z* axis a range of ±16,000 units, the *x* axis ±16,000 units, and *y* with ±16,000 units. Figure 12-8 shows bounding coordinates for motion in the positive axis of the world cube.

The coordinate system in this model is using fixed-point math. I chose this primarily because I am doing my development on a Dell 486/66 PC. In conversations with Direct3D developers at Microsoft, they recommended I use floating point over fixed point on a Pentium system. The Pentium processor can handle floating point operations a lot faster than fixed point. Due to my current system's lack of floating point horsepower, I will use fixed point math. The fixed point model I am using is from the *Black Art of 3D Programming*, by LaMothe. Listing 12-18 shows the data structures I use to maintain information on the virtual reality user's coordinates and displacements in our digital world.

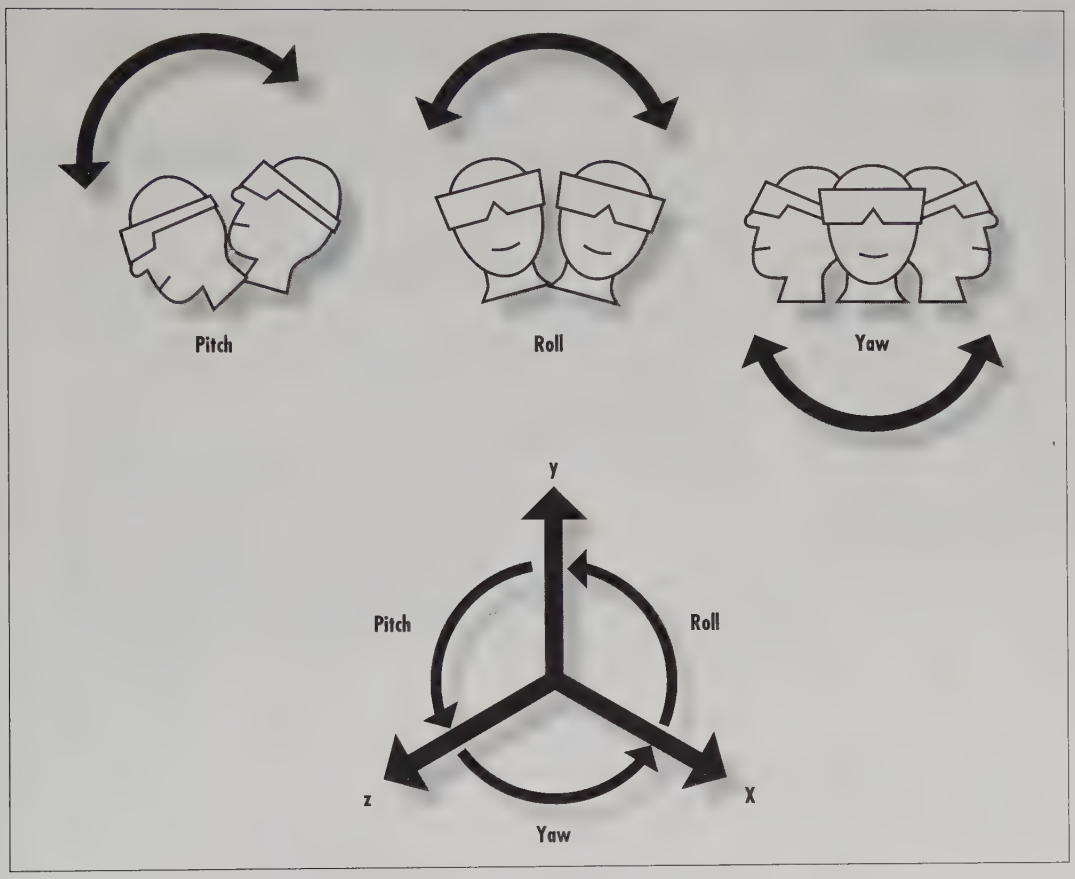


FIGURE 12-7 Headset orientations

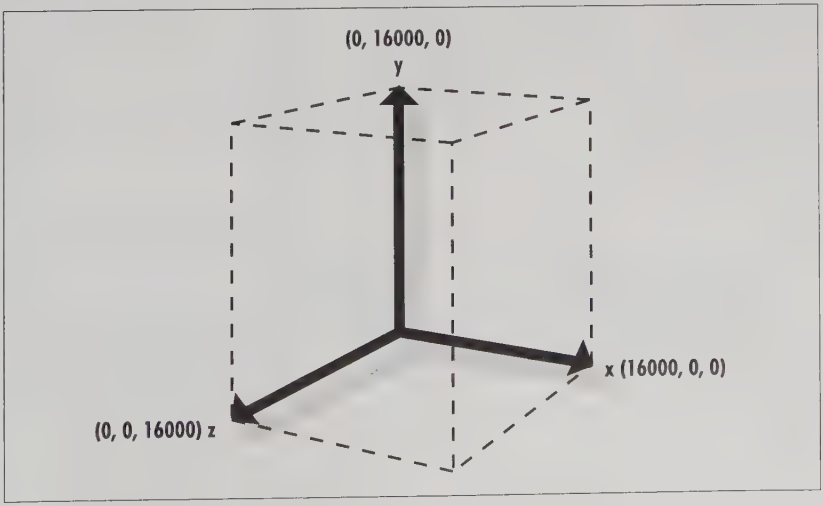


FIGURE 12-8 Positive right-handed coordinate system

LISTING 12-18 Data structures representing VR user orientation

```
typedef long fplong;     // Fixed Point is long
typedef long fpangle;    // Fixed Point Angle

typedef struct {
  fplong x;              // user x position
  fplong z;              // user z position
} XZPLANE, *PXZPLANE;

typedef struct {
  XZPLANE  heading;      // present heading
  XZPLANE  orientation;  // present orientation
  XZPLANE  location;     // present location xz plane
  fpangle  angle;        // present angle
  fplong   speedxz;      // speed in xz plane
  fplong   speedy;       // speed y
  fplong   viewery;      // viewer y
} VRINFO, *PVRINFO;
VRINFO vrUser;
```

When the Chapter 12 demos, *3DTomb* and DXMODEX, initialize, the virtual reality user (defined by the *vrUser* structure) is initialized to location $x = 0$ and $z = 0$. The view looking straight down the positive y axis is shown in Figure 12-9.

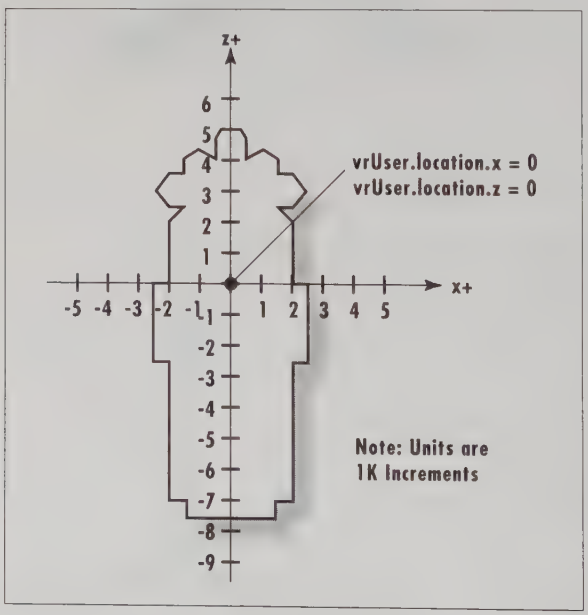


FIGURE 12-9
vrUser location on the *xz* plane

The player heading angle is zero, looking down the *xz* plane. The player heading, orientation, speed, and view angle are all initialized by the function InitWorld(), shown in Listing 12-19.

LISTING 12-19 Initializing the virtual world

```
BOOL initWorld( void )
{
  ddSurfaceWidth = CASTFPLONG(320/2);
  ddSurfaceHeight = CASTFPLONG(240/2);

  // read the BSP nodes.inf file
  if (ReadBSPTree() == FALSE)
    return FALSE;

  // set the location, direction, and speed
  vrUser.location.x = CASTFPLONG(0);
  vrUser.location.z = CASTFPLONG(0);
  vrUser.heading.x = CASTFPLONG(0);
  vrUser.heading.z = CASTFPLONG(1);
  vrUser.orientation = vrUser.heading;
  vrUser.speedy = CASTFPLONG(0);
  vrUser.viewery = CASTFPLONG(120);
  vrUser.speedxz = 0;

  return TRUE;
}
```

Linear updates in the virtual world are handled by the function UpdateVRUser(). This routine primarily handles and updates the displacements on the *xz* plane. The *xz* plane is bounded by ±16,000 units. The code for updating movement in the *xz* plane is shown in Listing 12-20.

LISTING 12-20 Linear updates on the *xz* plane

```
// 32000 x 32000 Cube Bounds
#define BOUNDS_MIN_X            (CASTFPLONG(-16000))
#define BOUNDS_MAX_X            (CASTFPLONG(16000))
#define BOUNDS_MIN_Z            (CASTFPLONG(-16000))
#define BOUNDS_MAX_Z            (CASTFPLONG(16000))

void UpdateVRUser( void )
{
  if (vrUser.speedxz != 0)
  {
    vrUser.location.x += fpMUL(vrUser.heading.x, vrUser.speedxz);
    if (vrUser.location.x <=BOUNDS_MIN_X)
      vrUser.location.x = BOUNDS_MIN_X;
    if (vrUser.location.x >=BOUNDS_MAX_X)
      vrUser.location.x = BOUNDS_MAX_X - 1;
```

continued on next page

continued from previous page

```
    vrUser.location.z += fpMUL(vrUser.heading.z, vrUser.speedxz);
    if (vrUser.location.z <=BOUNDS_MIN_Z)
      vrUser.location.z = BOUNDS_MIN_Z;
    if (vrUser.location.z >=BOUNDS_MAX_Z)
      vrUser.location.z = BOUNDS_MAX_Z - 1;
  }
}
```

Some complexities arise from using a VR headset. For instance, how to instruct a forward linear movement? I chose to use the keyboard or game controller for forward and backward motion responses. This would avoid awkward and unnatural motions by the user to use pitch displacements to move forward and backward.

Looking from side to side is a natural movement that can be easily implemented with a headset. Several methods are available for updating yaw. Whatever device is used for player response will still update the world in the same way. Listing 12-21 shows the simple mechanism used by the examples to update player angular changes.

LISTING 12-21 Updating yaw displacements

```
void UpdatePlayerRight( void )
{
  vrUser.angle -= ALPHA;
  if (vrUser.angle < CASTFPANGLE(0.0))
    vrUser.angle += CASTFPANGLE(360.0);
  vrUser.heading.x = fpCOS(vrUser.angle);
  vrUser.heading.z = fpSIN(vrUser.angle);
  vrUser.orientation = vrUser.heading;
}

void UpdatePlayerLeft( void )
{
  vrUser.angle += ALPHA;
  if (vrUser.angle > CASTFPANGLE(360.0))
    vrUser.angle -= CASTFPANGLE(360.0);
  vrUser.heading.x = fpCOS(vrUser.angle);
  vrUser.heading.z = fpSIN(vrUser.angle);
  vrUser.orientation = vrUser.heading;
}
```

This code can be used for the keyboard handler, game controller, or the headset. For details on how the GetKeyboard() and GetJoyStick() work, take a look at the sample code provided in Listing 12-21. Now to discuss the Virtual I/O i-glasses interface and how it is implemented.



Measuring YAW Displacement Using the Tracker

The source code for *3DTomb* uses the Virtual I/O library vstrack.lib, available to developers from Virtual I/O. The vstrack library has two data structures, TrackerData and TrackerStatus, that will be used for this example. The TrackerData td structure is passed to the library call readTracker(). This reads the Tracker's present *roll*, *pitch*, and *yaw* values. The call to requestTracker() sends the data to the application. The Tracker supports four different styles of data. The Euler values were recommended by Virtual I/O because they were device-independent. Listing 12-22 shows the code for retrieving the headset Tracker attitude values of *roll*, *pitch*, and *yaw*.

LISTING 12-22 Get the i-glasses Tracker angles

```
#include "vstrack.h"
TrackerData   td;
TrackerStatus ts;
long          trackerPresent=0;
float         yaw, pitch, roll;
float         refYaw, refPitch, refRoll;
float         dyaw, dpitch, droll;
float         oldYaw, oldPitch, oldRoll;
int           refSet;

long getTrackerAngles( float &yaw, float &pitch, float &roll)
{
  ts = readTracker(&td, timerTSecs(1));
  if (ts != TS_OK)
  {
       resetTracker(&td, timerTSecs(1));
       if (ts != TS_OK)
       {
      return 0;
      } // if
  } // if
  requestTrackerData(&td);
  yaw = TOFLOAT(td.euler.y);
  pitch = TOFLOAT(td.euler.x);
  roll = TOFLOAT(td.euler.z);
  return 1;
}
```

In the *3DTomb* example, the function HandleTracker() is responsible for updating user events via the i-glasses and Tracker. The human anatomy doesn't exactly move using smooth fluid motions as a robot would. For this reason, you have to filter the angles and information as they come in. Every 30 readings, swap the current reference readings of *roll*, *pitch*, and *yaw* with new ones. This provides a more stable and fluid movement for the user. Listing 12-23 shows how to filter values and update player angles using the i-glasses and Tracker.

LISTING 12-23 Updating using the Tracker

```
void resetAngles( void )
{
  refYaw = yaw;
  refPitch = pitch;
  refRoll = roll;
}
void HandleTracker( void )
{
  if (trackerPresent)
    getTrackerAngles(yaw,pitch,roll);

  if (refSet < 30)
  {
    refSet++;
       resetAngles();
  }
  dyaw = yaw - refYaw;
  dpitch = pitch - refPitch;
  droll = roll - refRoll;

  if (dyaw > oldYaw)
  {
    UpdatePlayerLeft();
  }

  if (dyaw < oldYaw)
  {
    UpdatePlayerRight();
  }

  oldYaw = dyaw;
}
```

Once the yaw displacement has been determined, then call UpdatePlayerRight() or UpdatePlayerLeft(). You'll notice that by using this, you have no need to keep the Tracker angles and the game angles in synch.



Other Considerations

I didn't add the pitch motion and handling to the game example because I didn't texture map the floors or the ceiling. There was simply nothing to look at. I want to return to this study and experiment with different texture-mapping techniques to convey the feeling of something monolithic or awesome in size. This was kind of hard to achieve using simple polygon fills, but it got my example up and running fast. Also, the walls lack solidity. This can be fixed by adding a check, using the dot product to determine which side of a line you're standing on. In this example, you can check it against the bottom edge of the polygon. No ray tracing is going on here; it's strictly a binary forest.

Before I go any further, I am giving in to the whims of Intel and upgrading to a Pentium. It's easier to debug computations when you're working in a natural number system rather than fixed point. Also, the BSP compiler from Michael Abrash's articles in *Doctor Dobb's* has its physical limitations, so I need to go off and write my own. I can't complain, though; Abrash's BSP compiler worked rather nicely for this 3D study. The *Black Art of 3D Game Programming* also has a discussion on BSP trees with some code examples.



Conclusion

Virtual reality is now at our fingertips. It has some flaws, but it's still in the pioneering stages. Based on what I have seen in the preparation of this chapter, the world is changing very quickly. I would almost say that joysticks may be on the way out by the end of the century. But then again, a new revolution in joysticks is taking place with force feedback technology.

Where is this technology headed? No one can say for sure. Will the momentum of this technology affect a society just as the arts and sciences moved the people of the Renaissance? I don't know. But I think for now, the European Renaissance artists still win hands down. I can still remember sitting in the Louvre in France looking at paintings. These 2D canvases of enormous size were able to draw you in. I haven't felt that sensation with stereoscopic glasses...yet.



INDEX

SYMBOLS

~ (tilde), 43
3D composition, 2D surfaces, 585
13h video mode, 108-110
20-bit addresses, *see* physical addresses
32-bit addresses, *see* IP (Internet Protocol)
48-bit addresses, *see* hardware addresses
8086 processors, 174

A

abstract data types, *see* ADTs
abstractions, 338
accessing DirectDraw interface using C++ (listing), 120
acknowledge messages, sending, 504
Activating SpaceFoil engines (listing), 329
activation flags, WM_ACTIVATEAPP, 369
active sockets, 474
Activision, 224
Adding a head and a tail object to cList class (listing), 340
Address Resolution Protocol (ARP), 469-470
addresses
  hardware addresses, mapping, 469-470
  IP addresses, 468-470
  IPX (Internet Packet eXchange), 471-472
  MAC (Media Access Control), 471
  physical addresses, 174
ADTs (abstract data types), 338
  insertions, 341-355
  removals, 345-374
  searches, 345, 347
Advanced Research Projects Agency (ARPA), 473
algorithms
  Bresenham line-drawing, 586
  Inorder Traversal, 360-362
  line-clipping, 301-306
allocating SLICE data structure, 434
Allocating and initializing pJoyData structure (listing), 212
Allocating LPWAVINFO memory (listing), 254
Amiga, 39
angles, deflection, 405-412
animating
  Bubonic the WartHog, 188-199
    keyboard processing techniques, 199-204
    mouse input messages, 204-210
  objects, 326-327
  sprites, 192-198
Animating giggling happy face, STEP structure (listing), 265
animation
  DIDRVR (DirectInput driver), 181-198
    CDib instantiation, 183-185
    DirectDraw, 186-188
    displays, 198
    sprite animation, 188-198
    terrain tiles, 190-192
  cells, loading, 183-186
  flicker, 62, 114-116
  objects
    CDib, 81-99
    speed, 328-330
    updating, 329-330
  offscreen surfaces, creating, 187-188

polygons, 316
sprites, updating, 189-199
tiles, 444-446
Animation for Bubonic with state HOG_GO_RIGHT (listing), 197
ANSI C, BSTs (binary search trees), 355-356
APIs (application programming interfaces), 124
DirectDraw, 123-137
DirectPlay, 476
WinG, 107
WinSock, 475
application buffering, 117
application layer, 467
applications
DIDRVR (DirectInput driver), 181-199
game application messaging, 25
generic game application, 19-24
"hello world" application, creating, 9
maximizing, 368-369
minimizing, 367-368
pausing, 369
terminating, 369-370
threads, concurrent, 371-374
windows, creating, 11-15
Windows 95, writing, 8-19
Approximating next x and y coordinates (listing), 588-590
ApproxNextX() function, 290, 294
ApproxNextY() function, 290
ApproxNextY() function (listing), 293
ApproxX() function, 587
ApproxY() function, 587
architectures
DIBs (Device-Independent Bitmaps), 40-41
memory
protected mode, 178-181
real mode, 174-178
arguments, passing, BITMAPINFOHEADER, 48
ARP (Address Resolution Protocol), 469-470
ARPA (Advanced Research Projects Agency), 473
arrays
gameObj, initializing, 384
goLeftRect, 197
goRightRect, 197
linked lists, compared, 337
polygons, drawing, 315
Shape class, moving objects, 322
standLeftRect, 197
standRightRect, 196
TerrainMap, 420-421
window boundaries, 304
assigning bounding rectangles, 396-402
Asteroid polygon definition (listing), 383
Asteroids game, 223, 384
collisions
angles of deflection, 405-412
detecting, 396-404
elastic, 404-405
velocity vectors, 388-389
disorders, creating, 387-388
levels, 386
Shape objects
defining, 384-385
drawing, 392
initializing, 385
processing, 393-394
velocities, determining, 388
Asteroids game function renderShapeList() (listing), 399-400
Attacking opponent player (listing), 554-555
attacking players, 553-555
attackPlayer() function, 553
attributes, players, 537-559
audio files
DirectSound, 227
IFF (Interchange File Format), 227
ILBM (IFF Interleaved Bitmap), 227
PCMWAVEFORMAT files, 235
RIFF (Resource Interchange File Format), 227-244
sound data structures, 252-267
WAVE, 235-237
WAVEFORMAT data structure, 233-235
WAVEINFO, 253-256

B

back flipping surfaces, tiles, drawing, 423
backgrounds, drawing, 422
bank-switched memory, video cards, 568
base tiles, initializing, 436-440
beginlevel values, slope equations, 426-427
beginThreadNT() function, 373
bfOffset field, 59
bfSize field, 59
biBitCount field, 48, 58
biClr field, 48
biClrUsed field, 58
Big Five Software, 223
binary search trees, see *BSTs*

binfoSize variable, 46
BIOS, keyboard interrupts, 177
_bios_keybrd() function, 178
BIOS-supported routines, 178
biPlanes field, 48
bIsActive variable, 365
BITMAPFILEHEADER (DIB), 40, 56
BITMAPINFO structure
  creating, 58
  DIB objects, 45
    copying, 65
    creating, 46
BITMAPINFOHEADER (DIB), 40-41
  DIB objects, 45
  fields, 47-49
  functions, creating, 48
  pointers, assigning values, 47
  reading, 57
  values, 48
bitmaps, code, converting to bottom-up, 63-65
biXPelsPerMeter field, 48
biYPelsPerMeter field, 48
BlitSurface() function, 191, 423
blitting, 104
  surfaces, DirectDraw, 145-152
  video drivers, 107
blocking, 475
Blt() function, 569
BltFast() function, 40, 183, 145, 273, 277
boundaries, windows
  line-clipping algorithms, 301
  setting, 304
  testing lines, 306-307
bounding rectangles
  creating, 396-397
  intersections, detecting, 402
Bresenham line-drawing algorithm, 283-284, 288-301, 547-553, 586
Bresenham Line-Drawing algorithm (listing), 297-299, 587
BresenhamEdge() function, 591
BresenhamLine() function, 587
broadcasts, 470
BSD UNIX, blocking, 475
BSP (Binary Space Partitioning) trees, 592
BST structure using C code (listing), 355
BSTs (binary search trees), 354-362
  defining, 355-356
  elements, inserting, 357-359
  empty, verifying, 358
  recursion, 357-363
Bubonic stops facing HOG_STOP_LEFT (listing), 197
Bubonic stops on WM_KEYDOWN message (listing), 201
Bubonic the WartHog
  animation, 188-199
    cell arrays, 192
    displaying, 198
    HOG_GO_RIGHT, 197
    HOG_STOP_RIGHT, 195-197
  backgrounds, drawing, 422
  base tiles, 430
  CLOUDS, 449
  collision detection routine, 458
  moving
    DirectInput, 210-219
    keyboard processing techniques, 199-204
    mouse input messages, 204-210
  slope, drawing, 428-430
  terrain maps, 420, 433
  tile CELL structure, 421
Bubonic's standing animation cell arrays (listing), 193
Bubonic's walking animation cell arrays (listing), 194
buffered input, DOS functions, 177-178
buffers
  DIBs (Device-Independent Bitmaps), 41
  DirectSound, creating, 246-247
  DWORD alignment, 50
  keyboard input, 177-180
  offscreen
    image bits, 62-63
    updating, 65-67
  sprites, 71
  updating, page flipping, 115
  video frames, writing pixels, 272
bypassing mouse, messaging, 208-210
  MS-DOS, INT 16h (interrupt 16h), 177-178

C

C++
  BSTs (binary search trees), 356
  CDib (DIB interface), 43-53
  classes
    cList, 338-341, 381-413
    cShapeListNode, 390
    cTree, 356-357
    nodes, 339
  operators, delete, 348

C++ Primer, 344
caches, ARP (Address Resolution Protocol), 470
calculating angles, theta angles, 406
CALLBACK functions, 477
  DlgProcHost(), 482-483
  DlgProcWhichGame(), 489-491
  DlgProvider(), 477
  GetDPServices(), 478
  JoinWhichGame(), 498-516
  prototypes, 478
Calling CreateSurface() using Spells of Fury (listing), 137
Calling the DirectSound member Play() (listing), 260
Calling the GetWAVRes() function (listing), 237
capturing mouse button input, 209
Case for dx < 0 (listing), 289
case 0: case statement, 496
CDib objects, initializing, 181-185
~CDib() member function, 43-45
CDib class, 43-53
  class scope operator, 44
  data members, 43
  member functions, 43-45
  objects
    animation, 81-99
    image bits, 51
    initializing, 66
  RECT data member, adding, 70
CDib class header file (listing), 52-53
CDib() member function, 43-45
CDib:: scope operator, 44
CELL databases, designing, 448-450
Cell occupation (listing), 543-544
CELL structures
  slope equations, 425
  tiles, 421
CELL structures declaration (listing), 524
cells
  arrays, animating, 192-195
  lookup tables, 426-427
  see also nodes
chaos, creating, 393-396
Characteristics of Bubonic the WartHog (listing), 188
CheckCollision() member function, 400-401, 403
CheckCollision() Shape class member function (listing), 400-401
Checking direction horizontal line (listing), 288
Checking if an element is in list (listing), 343
Checking keyboard state on demand (listing), 202
Checking state of NumLock key using DOS4/GW (listing), 177
Checking terrain map boundaries (listing), 543
checkKeyboard() function, 202-203, 367
children's games, Manhole, 261-267
Chunk pointer and data field declarations (listing), 239
chunks, RIFF files, 231-232, 237-242
classes
  CDib, 43-53
    data members, 43
    member functions, 43-45
    scope operator, 44
  cList, declaring, 339-340
  cListNode, 339
  cShapeListNode, 390
  cTree, 356-357
  polygons, creating, 319
  Shape, creating objects, 321-325
  WNDCLASS, 11-13
Classic C declaration for linked list (listing), 338
client/server communication, DirectPlay, messages, 502-516
clients
  DirectPlay, 485-489
  sockets, 474
clipping lines
  algorithms, 301-302
  calculating intersections, 310-312
  Cohen-Sutherland algorithm, 302
  game maps, 300-301
  intersections, 308-309
cList class, 381-413
  declaring, 339-340
  designing, 338-339
  drivers, 354
  head/tail objects, adding, 340-341
  modifying, 382-383
  nodes, inserting, 341-343
cList class (listing), 381-382
cListNode class, 339
CLOUDS
  designing, CELL databases, 448-450
  moving, 448-452
  sprite location and cell velocity listing, 449
clouds (networking), 466
code
  BITMAPINFO, calculating color table size, 46
  bitmaps, converting to bottom-up, 63-65

DDHAL (DirectDraw Hardware Abstraction Layer), 122
PowerSuits, 82-90
shapes, modifying, 323
code segment (CS) register, 174
Code used to check if tree is empty (listing), 358
Cohen-Sutherland line-clipping algorithm, 302-306
collision detections, lists, ITEMS, 457-459
collisions, 396
angles of deflection, 405-412
detection, 396-402
minimizing, 402-404
polygons, 412
elastic, 404-405
angles of deflection, 405-412
recoil, 406
velocity vectors, 407-409
multiple, 411-412
objects, inserting, 410-411
color, transparency, sprites, 75-78
color keys, palettes, DirectDraw, 155-166
color tables
DIB objects, calculating size, 46
initializing, 49
RGBQUAD
creating, 49
reading, 58-59
columns, tiles, slicing, 431-440
COM (Component Object Model), 118, 124
game environment compatibility, 118-122
interfaces, 119-121
Commodore, Amiga, 39
composing displays, DIDRVR, 198-219
compositeTransformation() function, 327
compression, RLE, 48
Computing new velocity components of space debris (listing), 409
Computing scanline offset from pVGAPtr (listing), 114
concurrent threads, 371-374
configuring
video drivers, memory, 109
palettes, DirectDraw, 152-155
connection-oriented architectures, 470
connection-oriented protocols, 470
connectionless architectures, 470-471
connectionless protocols, 470-472
console applications, Windows 95, 362
constructing lookup tables, slope, 427-428
Constructor for cShapeListNode class (listing), 390
constructors, 340
CDib(), 43-45
cShapeListNode class, 390
controllers, interrupt controllers, real mode, 174-175
converting coordinates to bottom-up images, 73
convex polygon scan conversions, 585-593
convex polygons, 313-314
Cooperative level and display mode gcType (listing), 572-573
cooperative levels
DirectDraw, 127-129
full-screen cooperative level, Mode X, 574-577
game console window, 572-574
coordinates
converting to bottom-up images, 73
DirectDrawSurface, memory, 273
display, DirectDrawSurface translations, 273-274
grids, calculating for sprite RECTS, 90-91
lines, defining, 279
polygons, rotation, 314
sprites, transparency, 76
world systems, game maps, 300-301
Copy BITMAPINFO function, 65
CopyDIBBits() function, 69, 74-75
calling arguments, 70
images, copying, 71
CopyDibBits() function, 64
CopyDibToSurface() function, 142, 188
copying
DIBs, 62-63
images, different buffer sizes, 71
sprites from DIB objects, 70
tables, RGBQUAD color table, 187-219
Copying CDib object image bits to DirectDraw surface using DC (listing), 143
CopyMemory() function, 250
CopyTransDIBBits() function, 78
Cosmic Osmo, 262
createDebris() function, 384
createLevel function initializes Shape list (listing), 386-387
createLevel() function, 385-387
CreatePlayer() member function, 501
CreateSoundBuffer() function, 247
CreateSurface() function, 134

CreateThread() function, 372
CreateWindow() function, 13
CreateWindowEx() function, 20, 23
Creating complex DirectDrawSurface using DDSCAPS structure (listing), 136
Creating DirectSound buffer (listing), 247
Creating DirectSound buffer from WAV resource (listing), 251
Creating DirectSound object (listing), 245
Creating node object (listing), 342
creating offscreen DIB (listing), 68-69
Creating offscreen DirectDrawSurface (listing)/Creating offscreen surface (listing), 142, 140, 187
Creating primary surface with attached back surface (listing), 136
Creating simultaneous sound buffers (listing), 255
Creating surfaces for a window (listing), 579
Creating window for Windows 95 (listing), 14-15
Critical game code in updateFrame() function (listing), 367
CS (code segment) register, 174
cShapeList and cShapeListNode class declarations (listing), 382-383
cShapeList class, 382-383
cShapeList member function RenderList() (listing), 392
cShapeListNode class, 382-383, 390
cTree class, declaring, 356-357
cTree member function InordertreeWalk() (listing), 361
cTree member function InsertTree() (listing), 358
currentCell field (wartHog struct), 189
CW_USEDEFAULT macro, 13

D

data members
  CDib class, 43
  lpbinfo, typecasting, 47
  LPRGBQUAD, 49
data structures
  BSTs (binary search trees), 354-362
  linked lists, 337-354
    C declaration, 338
    objects, 338-355
  SLICE, allocating, 434
Data structures representing VR user orientation (listing), 596-597
databases
  CELL, designing, 448-450
  SLICE data structure, reading tiles, 435
datagrams
  UDP (User Datagram Protocol), 471
    DirectPlay sessions, 494, 497-516
    DirectPlayer sessions, 501-502
DCs (device contexts), 51
  image bits, updating screens, 62-63
  images, drawing, 51
  screens, updating, 66
  sprites, drawing, 80-81
DDCOLORKEY struct (DirectDraw), 188
DDError() function, 126
DDHAL (DirectDraw Hardware Abstraction Layer), 122
DDInit() function, 186
ddraw.dll, 118
DDSInit() function, 138, 186
DDSURFACEDESC structure, 275
DDTOOLS function writeSurface() (listing), 276
DDTOOLS functions, 191, 198
DDTOOLS.CPP utility DDError() (listing), 127
Declaration for C++ cList class (listing), 340
Declaration of C++ node class (listing), 339
declaring classes
  CDib, 43
  cTree, 356-357
  node classes, 339
Defender, 223
defining
  BSTs, 355-356
  objects, shape objects, 383-390
  protocols, messages, 502-503
Defining a CELL in the terrain map (listing), 159
Defining a game GUID from the header file (listing), 484
Defining a location on a terrain map (listing), 159
Defining a polygon spaceship (listing), 315
Defining a spaceship from the point array (listing), 320
Defining simple protocol messages (listing), 503
Defining the GameSessionsDefs structure (listing), 484
Defining the terrain map (listing), 161
Definition of the point structure (listing), 314
deflection angles, 405-412
DefWindowProc() function, 18
delay field (wartHog struct), 189
delete operator, 348
deleting objects, 347-348
Deleting the item once it's found (listing), 347
deltax values, 206
Deluxe Paint, 39

Describing offscreen DirectDrawSurface using DDSURFACEDESC (listing), 140
descriptors
IDT (interrupt descriptor table), 178
sockets, 474
Design for a C++ list class (listing), 339
designing
C++ classes, 338-341
CELL databases, 448-450
game process scheduler, 365
destructors, 43-45, 340
detecting collisions, 396-404
determining
recoil constants, 406
vectors
direction, 406
velocity, 388, 406-413
Determining asteroid velocities (listing), 389
Determining if buffer is in use (listing), 258
Determining if DirectDrawSurface is lost (listing), 144
device contexts, *see* DCs
device drivers, 116-119
Device-Independent Bitmaps, *see* DIBs
DGNPlayerHitNotify() function, 555
DGNPlayerPosNotify() function, 544
DGNReadPosNotify() function, 545-546
Dialog box to join active DirectPlay session (listing), 498
dialog boxes, 30-33
DlgProc() function, 30-32
DPTOOLS.RC, 491
drivers, 32-33
resources
ID IDD_CHOOSEPROVIDER, 477
IDD_WHICHGAME, 491
Dialog controls for selecting the session provider (listing), 479
Dialog to host or join a game session (listing), 483
DialogBox() function, 32
DIB objects
BITMAPINFO structure, creating, 46
creating, 45-53
image bits
DWORD alignment, 50
updating screens, 62-63
memory, 45
offscreen, creating, 65-69
reading, 54
sprites
copying, 70
transparency, 76
DIBs (Device-Independent Bitmaps), 37-99
architecture, 40-41
blitting, 104
copying, 62-63
system memory, 41
DIDRVR
keyboard input, VK_NUMPAD6 key, 201-204
mouse input messages, 204-210
DIDRVR (DirectInput driver), 181-199
CDib instantiation, 183-185
DirectDraw, 186-188
displays, 198
sprite animation, 188-198
terrain tiles, 190-192
DIGENX.CPP file, 198
DIGENX.CPP source file, 207
DIGLBLS.CPP file, 192
Direct X, applications, NetChess, 508-511
Direct X Game SDK, 566
DirectDraw, 123
API, 124-157
cooperative levels, 127-129
DDCOLORKEY struct, 188
DirectDrawPalette, creating, 187-188
DirectDrawSurface, creating, 188
game console window, 572-574
interface, 118
Mode X
display settings, 574-577
flipping surfaces, 577-584
full-screen cooperative level, 574-577
initializing, 571-577
surface creation, 578
video cards, 567-569
objects
creating, 125-127, 186
initializing, 124-132
palettes, 49, 152-157
color keys, 155-166
setting, 152-155
protected mode addressing, 106
surfaces, 133-152, 450-451
blitting, 145-152
flipping, 145-152

initializing, 135
offscreen surface, 140-152
primary surface, 134-139
restoring, 143-144
video cards, memory architecture, 567-569
video modes, 129-134
DirectDraw method GetAttachedSurface() (listing), 138
DirectDraw object instantiation (listing), 126
DirectDraw objects
DirectDrawSurface video frame buffers, 272
DirectDrawClipper object, 580
DirectDrawCreate() function, 125
DirectDrawSurface, 272, 450-451
accessing functions, 275
display coordinates, translating, 273-274
memory location coordinates, 273
pixels, writing, 276
slices, drawing, 442-443
DirectInput, 210-218
device input, processing, 214-219
joystick driver, initializing, 210-219
DirectInput driver, *see* DIDRVR
directional vectors, filled polygons, 585
DirectPlay, 472-475
API (application programming interface), 476
blocking, 475
client/server dialog box, creating, 482-483
clients, initializing, 488-489
datagrams, 494, 497-516
DPSESSIONDESC structure, initializing, 496
drivers, creating, 499-516
EnumSessions list box, initializing, 491
GameSessionsDefs data structure, 484-486
GUIDs (Globally Unique Identifiers), 483-484
messages
processing, 506
reading, 505-506, 508
sending, 503-504
structure, 502
objects, creating, 476
ports, 472-473
servers
initializing, 486-488
tournament servers, 508-511
sessions
client/server communication, 502
initializing, 476-482
joining, 489-491
opening, 497
session providers, 477-481
WinSock, 494
sockets, 473-475
DirectPlayCreate() function, 480
DirectPlayEnumerate() function, 477-478
DirectPlayer, 501-502
DirectSound, 227
buffers, creating, 246-247
interface, 244-261
objects, creating, 244-252
sound data structures, 252-267
WAVEINFO data, playing, 256-267
DirectSoundCreate() function, 244
DispatchMessage() function, 16, 29
display modes, game console window, 572-574
display settings, Mode X, 574-577
displaying generic game applications, 60-62
displays, DIDRVR, composing, 198
distance translation, polygons, 316-317
distributed environments, player movements, 544-546
DlgProc() function, 30-32
DlgProcHost driver function GetGameHost() (listing), 483
DlgProcHost() function, 482-483
DlgProcWhichGame() function, 489-491, 498
DlgProvider() function, 477
DOS
13h video mode, 108-110
environments, protected mode, 105
extenders, 105-106
games, frame-rate requirements, 105
video architecture, 108
video mode X, 110
DOS extenders, *see* protected mode
DOS functions, buffered input, 177-178
DOS4/GW, 105, 107
DPID (DirectPlay ID), 501
DPOPEN_OPENSESSION flag, 496
DPRecv() function, 507-508
DPSESSIONDESC structure, initializing, 496
DPTOOLS functions
InitDPServer(), 487
JoinWhichGame(), 489
SessionDriver(), 499-500
DPTOOLS.RC dialog box, 477, 491
drawBackGround( ) function, 451-452
DrawBits() function, 51, 143
drawBubonic() function, 195, 367, 428-429

drawCells driver, TERRAIN program, 447
drawCells( ) function, 441, 445-447, 548, 553
DrawCells() function, 550
drawDisplay() function, 390, 392-413
drawing
  backgrounds, 422
  Bubonic the WartHog, slope, 428-430
  filled polygons, 590-592
  Generic Game Application screen, 51-52
  ITEMS lists, 456-457
  lines, 278
    Bresenham algorithm, 283-284
    determining primary axis, 283-286
    equations, 282
    error terms, 283-284
    slope, 279-281
    x pixel approximation, 294-295
    y coordinate loops, 295-299
    y pixel approximation, 290-293
  polygons, 314-316, 590-592
  slices, 440-443
  sprites, DCs, 80-81
  terrain tiles, 191-199
  tiles, 423, 443
  see also rendering
Drawing a filled polygon (listing), 590-591
Drawing Bubonic in any of his motion states (listing), 194
Drawing cells Bresenham algorithm (listing), 550-553
Drawing the array to the back surface (listing), 591-592
Drawing the DIDRVR display (listing), 198
Drawing the same cell for n milliseconds (listing), 196
Drawing the Shape objects for Asteroids (listing), 392
Drawing the SpaceFoil explosion (listing), 332
Drawing the terrain tiles (listing), 191
Drawing tiles from lpDDSOne to lpDDSBack (listing), 165
drawItems() function, 457
drawOneToBack() function, 197, 422, 526
drawScanLineList() function, 591
drawTerrain() function, 191, 367, 422-423
drivers
  devices, 116-119
  dialog boxes, 32-33
  DirectPlay, creating, 499-516
  drawCells, TERRAIN program, 447
  video, 107-109
DSDriver() function, 261, 263
DSInit() function, 244
DSTOOLS, Xplore games, 262-263
Dungeon
  PLAYER linked list, 529-533
  players, 528-538
    attacking, 553-555
    attributes, 547-559
    initializing attributes, 537-538
  potions, 555-557
  terrain map, 521-528
    horizontal scrolling terrain maps, 521-528
    initializing, 524-526
    locating tiles, 526-528
    player movements, 538-547
    vertical scrolling terrain maps, 521-528
Dungeon DirectPlay messaging structure (listing), 544
Dungeon terrain map (listing), 522-523
dwFlags parameter, 372
DWORD alignment, image bits, DIB objects, 50

E

elastic collisions, 404-405
  angles of deflection, 405-412
  recoil, 406
  velocity vectors, 407-409
Electronic Arts Interleaved Bitmap, *see* ILBM
Eliminator, Big Five software, 223
Emulating a display mode in window (listing), 574
encapsulation, 468
encode() function, 305
encodeObject() function, 320
encodeSpriteRect() function, 397
encodeSpriteRect() Shape class member function (listing), 397
Encoding the logic arrays (listing), 305
EndDialog() function, 483
endlevel values, slope equations, 426-427
endpoints, swapping, 288
Enumerated list of game shape (listing), 323
enumerated lists, active game sessions, requesting, 495-496
EnumSessions list box, initializing, 491
EnumSessions() member function, 492
environments
  DOS, protected mode, 105
  games, compatibility, 118-122
equations
  lines, drawing, 282
  slope, 425-427

EraseBackSurface() function, 186, 198
ErasePrimarySurface() function, 186
error terms, lines, drawing, 283-284
errors, GPF (General Protection Fault), 179
Ethernet, IEEE 802.3 standard, 469
Ethernet networks, 465-467
event messages, Windows 95, 180
events
    * key, 199
    WM_KEYDOWN, 199-202
Example of a list object for a mechanized robot battalion (listing), 352
Examples of page scripts used in the Xplore game (listing), 264
extenders, 105-106
external interrupts, real mode, 174

F

fgetpos( ) function, 433
fields
    bfOffset, 59
    bfSize, 59
    biBitCount, 48, 58
    biClr, 48
    biClrUsed, 58
    biPlanes, 48
    BITMAPINFOHEADER, 47-49
    biXPelsPerMeter, 48
    biYPelsPerMeter, 48
    pwrSuitPosX, 86
    pwrSuitPosY, 86
file extensions, 43
File Transfer Protocol (FTP), 467
files
    audio files
        DirectSound, 227
        IFF (Interchange File Format), 227
        ILBM (IFF Interleaved Bitmap), 227
        PCMWAVEFORMAT files, 235
        RIFF (Resource Interchange File Format), 227-233
        WAVE, 235-237
        WAVEFORMAT data structure, 233-235
        WAVEINFO, 253-256
    .BMP, 40, 183
    ddraw.dll, 118
    DIGENX.CPP, 198
    DIGLBLS.CPP, 192
    DPTOOLS.RC, 477
    extensions, 43
    format chunks, unpacking, 242-243
    GENX.CPP, 181
    header files, 43-44
        STDIO.H, 9
        WINDOWS.H, 9, 200
    IFF (Interchange Format File), 228-231
    RIFF, 231-233
    source files, DIGENX.CPP, 207
    TERRAIN.INF
        initSlices( ) function, 439-440
        opening, 432-433
        SLICE data structure, 435
    WAVE files, 235-237
filled polygons, 584-593
    2D surfaces, 3D composition, 585
    BSP (Binary Space Partitioning) trees, 592
    convex polygon scan conversions, 585-593
    directional vectors, 585
    drawing, 590-592
fillPlayerInfo() function, 531, 533
FindResource() function, 236
flags, DPOPEN_OPENSESSION, 496
flicker, page flipping, 114-116
Flip() function, 145
flipping surfaces, DirectDraw, 145-152, 577-584
flipSurface() function, 198, 367
floodFillPolygon() function, 590
format chunks, unpacking, 242-243
forms, RIFF, 232-233
Four Layer Network Model, 467
Four-byte chunk ID declarations (listing), 230
fragmentation, IP (Internet Protocol), 471
frames, referencing, virtual reality, 594-598
FreeCDib() member function, 44-45
Freeing the Spells of Fury LPWAVINFO buffer (listing), 256
fscanf() function, 435
fseek() function, 433
FTP (File Transfer Protocol), 467
full-screen cooperative level, Mode X, 574-577
functions
    _bios_keybrd(), 178
    ApproxNextX(), 290, 294
    ApproxNextY(), 290
    ApproxX(), 587
    ApproxY(), 587

attackPlayer(), 553
beginThreadNT(), 373
BlitSurface(), 191, 423
Blt(), 569
BltFast(), 145, 273, 277
BresenhamEdge(), 591
BresenhamLine(), 587
checkKeyboard(), 202-203, 367
compositeTransformation(), 327
Copy BITMAPINFO(), 65
CopyDIBBits(), 69-75
CopyDibBits(), 64
CopyDibToSurface(), 142, 188
CopyMemory(), 250
CopyTransDIBBits(), 78
createDebris(), 384
createLevel(), 385-387
CreateSoundBuffer(), 247
CreateSurface(), 134
CreateThread(), 372
CreateWindow(), 13
CreateWindowEx(), 20, 23
creating, BITMAPINFOHEADER, 48
DDError(), 126
DDInit(), 186
DDSInit(), 138, 186
DefWindowProc(), 18
DGNPlayerHitNotify(), 555
DGNPlayerPosNotify(), 544
DGNReadPosNotify(), 545-546
DialogBox(), 32
DirectDrawCreate(), 125
DirectDrawSurface, accessing, 275
DirectPlayCreate(), 480
DirectPlayEnumerate(), 477-478
DirectSoundCreate(), 244
DispatchMessage(), 16, 29
DlgProc(), 30-32
DlgProcHost(), 482-483
DlgProcWhichGame(), 489-491, 498
DlgProvider(), 477
DPRecv(), 507-508
drawBackGround(), 451-452
DrawBits(), 51, 143
drawBubonic(), 195, 367, 428-429
drawCells(), 441, 445-447, 548, 553
drawDisplay(), 390, 392-413
drawItems(), 457
drawOneToBack(), 197, 526
drawScanLineList(), 591
drawTerrain(), 191, 367, 422-423
DSDriver(), 261, 263
DSInit(), 244
encode(), 305
encodeObject(), 320
EndDialog(), 483
EraseBackSurface(), 186, 198
ErasePrimarySurface(), 186
fgetpos(), 433
fillPlayerInfo(), 531, 533
FindResource(), 236
Flip(), 145
flipSurface(), 198, 367
floodFillPolygon(), 590
fscanf(), 435
fseek(), 433
GameMsg(), 506-516
GetAsyncKeyState(), 202-204, 209
GetAvailableBuf(), 259
GetCaps(), 569, 578
GetCDibBiHeight(), 142
GetCDibBiWidth(), 142
GetCursorPos(), 208
GetDC(), 143
GetDlgItem(), 478
GetDPServices(), 478
GetGameHost(), 483, 500
gethostbyname(), 475
GetJoyStick(), 599
getJoyStick(), 214-215
GetKeyboard(), 599
GetMessage(), 27-28, 51, 363
getMouse(), 208-219
GetNewSurface(), 141, 188, 579
GetPixelByte(), 77
GetProvider(), 500
getShapeAngle(), 326
GetStatus(), 258
GetWAVRes(), 236, 247
handleMouse(), 205-206
HandleTracker(), 599
image bits, reading, 59
InitCDibObjects(), 66
InitDITest(), 183, 185
InitDPServer(), 487
InitDXModex(), 576
initGame(), 386
InitGameApp(), 20, 66
InitGameDefs(), 486-487
initGameObject(), 385

initItems(), 454
initJoyThread(), 373
initObject(), 325
initPlayer(), 531
initSlices(), 439-440
initSpaceFoil(), 324
InitWAVInfo(), 263
InitWorld(), 597
IsWindowVisible(), 368
JoinWhichGame(), 489, 498-516
joyGetDevCaps(), 213
joyGetNumDevs(), 212
joyGetPosEx(), 215
lineOfSight(), 549-550
LoadResource(), 236
Lock(), 249, 275
lread(), 58
main(), 8
MakeSoundBuffer(), 251
memset(), 50
moveObject(), 329
moveShape(), 329
MoveSuit(), 86
NCMontyHall(), 515
NCMoveCompleteNotify(), 514
NCReadMove(), 514-515
NCReadTurn(), 513
NCTurnNotify(), 511
newItem(), 455
NumberOfColorEntries(), 58
objectTransformation(), 327
PeekMessage(), 28, 363
pickUpItem(), 556
playerMove(), 541
PostQuitMessage(), 370
processGame(), 390-398, 411-412
QueryInterface(), 121
rand(), 387
randNumGen(), 387
ReadFile(), 54
ReadHelloMsg(), 507
readTracker(), 599
RealizePalette(), 117
Receive(), 476
recursive, 357
recv(), 475
RegisterClass(), 11
reject(), 307
releasePiece(), 514
renderShapeList(), 399-400
requestTracker(), 599
Restore(), 144
rotateObjectLeft(), 326, 393
rotateObjectRight(), 393
rotateShape(), 320
rotateSpaceFoilLeft(), 326
Send(), 476, 545
SessionDriver(), 499-500
SetCDibState(), 56
SetCooperativeLevel(), 127, 129
SetCursorPos(), 206
setDebrisVectors(), 406-409
SetDisplayMode(), 129
setDisplayMode(), 119
SetFocus(), 22
SetPalette(), 153
SetRGBQuadPalette(), 187
setScanLineEntry(), 590
SetTransColorKey(), 188
SetTransparentColor(), 157
sfCheckCollision(), 403, 412
Shape::encodeObject(), 325
Shape::setAngle(), 325
ShowCursor(), 22
ShowWindow(), 14, 368
socket(), 474
StretchDIBits(), 51
swapTermsInt(), 288
thruster(), 328
TranslateMessage(), 16, 29
trivialAccept(), 306-307
tumbleDebris(), 395
Unlock(), 251, 275
UnpackWAVChunk(), 247
updateDebrisCollision(), 403-409
updateFrame(), 198, 207, 367, 424
UpdatePlayerLeft(), 600
UpdatePlayerRight(), 600
UpdateScreen(), 66
UpdateVRUser(), 597
UpdateWindow(), 15
WinFunc(), 67
WinMain, 198
WinMain(), 8-11, 15, 22-23, 531
WndFunc(), 12, 16-19, 199-200, 204
writeSurface(), 276
WSAAsyncGetHostByName(), 475
WSAIsBlocking(), 475
see also member functions
Functions used for selecting the session provider (listing), 480

G

game console, updating, 583-584
game console window
  cooperative levels, 572-574
  creating, 19-30
  display modes, 572-574
  initializing, 20-22
Game host server (listing), 510
game maps, 300-302
game message loops, 27-28
game messages, requesting, 495-496
game process scheduler, designing, 365
Game process scheduler (listing), 365
Game SDK, 118
GameMsg() function, 506-516
gameObj array, 384-385
games
  children's games, Manhole, 261-267
  DOS, frame-rate requirements, 105
  environments, compatibility, 118-122
  events, scheduling, 364-371
  levels, initializing, 386
  messaging, processing, 25
  pausing, 369
  RPGs (role-playing games), 519
  terminating, 369-370
GameSessionsDefs data structure, 484-486
gateways, *see* routers
GDI (Graphics Device Interface), 116-118, 124
General Protection Fault (GPF) errors, 179
generic game applications, 19-24
  DIB objects, creating offscreen, 65-67
  displaying, 60-62
  screen, drawing, 51-52
Generic Game Application's message loop (listing), 148
generic game application's WinFunc (listing), 23
generic game application's WinMain() (listing), 22-23
GENX.CPP file, 181
Get the i-glasses Tracker angles (listing), 599
GetAsyncKeyState() function, 202-204, 209
GetAvailableBuf() function, 259
GetCaps() function, 569, 578
GetCDibBiHeight() function, 142
GetCDibBiWidth() function, 142
GetCursorPos() function, 208
GetDC() function, 143
GetDlgItem() function, 478
GetDPServices() function, 478
GetGameHost() function, 483, 500
gethostbyname() function, 475
GetJoyStick() function, 599
getJoyStick() function, 214-215
getJoystick() function (listing), 214
GetKeyboard() function, 599
GetMessage() function, 27-28, 51, 363
getMouse() function, 208-209
GetNewSurface() function, 141, 188, 579
GetPixelByte() function, 77
GetProvider() function, 500
getShapeAngle() function, 326
GetStatus() function, 258
Getting mouse updates on demand (listing), 208
Getting up and running, Spells of Fury, DDTOOLS and Cdib object (listing), 157
GetWAVRes() function, 236, 247
GlobalAllocPtr() operator, 45
GlobalFreePtr() operator, 45
Globally Unique Identifiers, see *GUIDs*
goLeftRect array, 197
goodPowerSuit structure, 86
goRightRect array, 197
GPF (General Protection Fault) errors, 179
graphics formats, 39
grids, coordinates, calculating for sprite RECTS, 90-91
GUIDs (Globally Unique Identifiers), 121, 483-484

H

.h file extension, 43
handleMouse() function, 205-206
handlers, interrupt handlers, 175-178
HandleTracker() function, 599
Handling collision with shape of type BIG_ROCK_INDEX (listing), 411
handling messaging, 28-29
Handling the mouse WM_LBUTTONDOWN message (listing), 205
Handling the WM_KEYDOWN event for VK_NUMPAD6 (listing), 201
hardware addresses, mapping, 469-470
head objects, declaring, 340-341
header files, 43-44
  shapes, modifying code, 323
  STDIO.H, 9
  WINDOWS.H, 9
headsets, virtual reality, 593-601

health, players, RPGs, 553-559
hello messages
  reading, 507
  sending, 504
"hello world" application, Windows 95, creating, 9
HIWORD macros, 205
HIWORD() macro, 26
HOG_GO_RIGHT (wartHog struct), 197
HOG_STOP_LEFT (wartHog struct), 196-197
HOG_STOP_RIGHT (wartHog struct), 195-196
horizontal scrolling terrain maps, 521-528
.hpp file extension, 43

I

ID IDD_CHOOSEPROVIDER, 477
IDirectDraw COM interface (listing), 120-121
IDs
  datagrams, DirectPlay, 501
  ports, 472-473
IDTs (interrupt descriptor tables), 178
IEEE 802.3 standard, 469
IF (interrupt flag), 174
if statements, copy loops, transparency values, 77
IFF (Interchange File Format), 227-231
IFF BitMapHeader structure (listing), 230
IIDs (Interface Identifiers), 121
ILBM (IFF Interleaved Bitmap), 227
image bits
  buffers, offscreen, 62-63
  DIB objects, DWORD alignment, 50
  pointers, 64, 72
  reading, 59
  size, 59
image bits memory, allocating, 50
images
  converting coordinates, 73
  copying different buffer sizes, 71-73
  locations, describing, 70
  pixels, writing to DirectDrawSurface, 272
  rows, indexes, 76
  scanlines, offset values, 76
imitating
  elastic collisions, 404-405
    angles of deflection, 405-412
    recoil, 406
    velocity vectors, 407-409
Implementing the best fit allocation scheme (listing), 582-583
indexes, images, rows, 76
InitCDibObjects() function, 66
InitDIBObjects transparency blitting (listing), 80-81
InitDITest() function, 183, 185
InitDPServer() function, 487
InitDXModex() function, 576
initGame() function, 386
InitGameApp() function, 20, 66
InitGameDefs() function, 486-487
initGameObject() function, 385
initializing
  arrays, gameObj, 384-385
  CDib objects, 181-185
  color tables, 49
  DirectPlay
    clients, 488-489
    DPSESSIONDESC structure, 496
    EnumSessions list box, 491
    servers, 486-488
    session providers, 477-481
    sessions, 476
  game console window, 20-22
  games, levels, 386
  joystick drivers, DirectInput, 211-219
  lists, ITEMS, 453-456
  Mode X, DirectDraw, 571-577
  nodes, DirectPlay, 482-483
  objects, 324
    CDib, 66
    DirectDraw, 124-132
    pDibBckGrnd, 66
    Shape objects, 385
  player attributes, 537-538
  PLAYER linked list (Dungeon), 530-533
  PLAYER node, 533-537
  pointers, 346
  structures, 212
  surfaces, DirectDraw, 135
  terrain maps, 524-526
  tiles, 436-440
  velocity vectors, 407
Initializing DirectDraw in the DIDRVR application (listing), 186
Initializing DirectDrawSurface with DDSInit() routine (listing), 139
Initializing a level in Asteroids (listing), 386
Initializing a Shape object (listing), 325

Initializing a shape object array (listing), 324
Initializing afterburner Shape objects (listing), 325
Initializing DIDRVR application (listings), 182
Initializing DirectDraw for 640 x 480 resolution (listing), 131
Initializing DirectDraw for Mode X (listing), 576-577
Initializing DirectInput (listing), 211
Initializing DirectDrawClipper object (listing), 580
Initializing DirectPlay game server (listing), 487
Initializing DPSESSIONDESC to connect to session (listing), 496
Initializing Dungeon PLAYER linked list (listing), 530-531
Initializing encoding logic arrays (listing), 305
Initializing EnumSessions list box (listing), 491
Initializing game shape objects (listing), 385
Initializing gameObj shape array (listing), 385
Initializing GameSessionDef structure for game server (listing), 486
Initializing GameSessionDefs structure for client and server (listing), 485
Initializing GameSessionDefs structure for game client (listing), 489
Initializing generic game application window (listing), 21
Initializing PLAYER attributes (listing), 538
Initializing PLAYER cell positions (listing), 537
Initializing PLAYER fields (listing), 535-536
Initializing session provider dialog (listing), 478
Initializing spaceship Shape objects (listing), 324
Initializing terrain map (listing), 162
Initializing values in the GameSessionDefs (listing), 484
Initializing velocity vectors of space debris (listing), 407
Initializing virtual world (listing), 597
initItems() function, 454
initJoyThread() function, 373
initObject() function, 325
initPlayer() function, 531
initSlices() function, 439-440
initSpaceFoil() function, 324
InitWAVInfo() function, 263
InitWorld() function, 597
Inorder Traversal algorithm, 360-362
InorderTreeWalk() member function, 357, 361-374
input, keyboards, WM_KEYDOWN event, 199-204
inserting
  nodes, 341-343
  Shape objects, 410-411
Inserting a PLAYER into an empty list (listing), 531
Inserting an element into the tree (listing), 359
Inserting into the cList object (listing), 341
Inserting PLAYER into nonempty list (listing), 533
InsertList() member function, 341
InsertTree() member function, 357-358
instruction pointer (IP) register, 175
INT 16h (interrupt 16h), 177
INT 21h (interrupt 21h), 177
Intel 8086 processors, 174
Interface Identifiers, *see* IIDs
interfaces, 171
  COM, 120-121
  DIB (Device-Independent Bitmap), 42-53
  DirectDraw, 118
  DirectSound, 244-261
Internet messaging, 470-471
Internet Packet eXchange (IPX), 471-472
Internet Protocol, *see* IP
interrupt 16h (INT 16h), 177
interrupt 21h (INT 21h), 177
interrupt descriptor tables (IDTs), 178
interrupt flag (IF), 174
interrupts
  protected mode, 178
  real mode, 174-175
    controllers, 174-175
    DOS functions, 176-177
    handlers, 175-178
    keyboard interrupts, 177
    vector tables, 175-176
IP (Internet Protocol), 468-471
IP (instruction pointer) register, 175
IPX (Internet Packet eXchange), 471-472
IPX/SPX, 471
IRET, 176
IsWindowVisible() function, 368
ItemCalls lookup table, 455
ITEMS lists, 452-453
  collision detections, 457-459
  drawing, 456-457
  initializing, 453-456
iteration (lists), 390-393
IUnknown COM interface, 121
IUnknown COM interface (listing), 121

J-K

joining DirectPlay sessions, 489-491
Joining an active game session (listing), 489
JoinWhichGame() function, 489, 498-516
JOYCAPS struct, 213
JOYDATA struct, 218
JOYDATA structure, 211-212
joyGetDevCaps() function, 213
joyGetNumDevs() function, 212
joyGetPosEx() function, 215
joysticks
  buttons, drivers, 218
  input, checking, 214-216
  interfaces, DirectInput, 210-218

* key event, 199
keyboards
  input, 199-204
  keys, virtual keys, 200
  user input, checking, 202-219

L

lag time, minimizing, concurrent threads, 371-372
LANs (local area networks), 465-467
lastTickCount field (wartHog struct), 190
Launching a thread using the scheduler (listing), 371
Lawnmower Man, 594
layers (networking), 467
  application, 467
  link, 469
  network, 468
  protected mode environments, 105
  protection, PL 0 (protection layer 0), 179
  transport, 468
LB_GETITEMDATA message, 496
LBN_DBLCLK message, 480
levels (games), initializing, 386
Linear updates on the xz plane (listing), 597-598
lineOfSight() function, 549-550
lines
  clipping
    algorithms, 301-302
    calculating intersections, 310-312
    Cohen-Sutherland algorithm, 302
    encoding endpoints, 304-306
    game maps, 300-301
    intersections, 308-309
  defining, coordinates, 279
  drawing, 278
    Bresenham algorithm, 283-284
    determining primary axis, 283-286
    equations, 282
    error terms, 283-284
    slope, 279-281
    x pixel approximation, 294-295
    y coordinate loops, 295-299
    y pixel approximation, 290-293
    y-intercept, 281-282
  orientation, checking, 285-286
  Point-Slope equation, 281-282
  windows boundaries, testing, 306-307
link layer, 469
linked lists, 337-354
  arrays, compared, 337
  list objects, 338-341, 352-353
  nodes
    inserting, 341-343
    removing, 343-351
  searching, 346-347
Linux, 486
list objects
  finding, 343-344
  gaming applications, 352-353
  processing, 390-393
listings
  Accessing DirectDraw interface using C++, 120
  Activating SpaceFoil engines, 329
  Adding head and tail object to the cList class, 340-341
  Allocating and initializing the pJoyData, 213
  Allocating LPWAVINFO memory, 254
  Animating a giggling happy face, 266-267
  Animation for Bubonic, 197-219
  animations, 444
  Approximating next x and y coordinates, 588-590
  ApproxNextY() function, 293
  Asteroid polygon definition, 383
  Asteroids game function renderShapeList(), 399-400
  Attacking opponent player, 554-555
  Bresenham Line-Drawing algorithm, 297-299, 587
  BST structure using C code, 355-356
  Bubonic stops facing HOG_STOP_LEFT, 197
  Bubonic stops on WM_KEYDOWN message, 201-202

Bubonic the WartHog
base tiles, 430
CLOUDS, 449
collision detection routine, 458
drawing backgrounds, 422
terrain maps, 420, 433
tile CELL structure, 421
Bubonic's standing animation cell arrays, 193-219
Bubonic's walking animation cell arrays, 194
Calling CreateSurface(), 137
Calling the DirectSound member Play(), 260-261
Calling the GetWAVRes() function, 237
Case for dx < 0, 289
Case for dx > 0, 289
CDib class header file, 52-53
CELL lookup table for cloud sprites, 448
Cell occupation, 543-544
CELL structure declaration, 524
Characteristics of Bubonic the WartHog, 188-189
CheckCollision() Shape class member function, 400-401
Checking direction of a horizontal line, 288
Checking keyboard state on demand, 202-203
Checking state of the NumLock key, 177
Checking terrain map boundaries, 543
Checking to see if element is in the list, 343-344
Chunk pointer and data field declarations, 239
Classic C declaration for a linked list, 338
cList class, 381-382
CLOUDS sprite location and cell velocity, 449
Code used to check if tree is empty, 358
Cohen-Sutherland line-clipping algorithm, 311-332
Computing a scanline offset, 114
Computing new velocity components of space debris, 409
Constructor for cShapeListNode class, 390
Cooperative level and display mode gcType, 572-573
Copying CDib object image bits, 143
createLevel function initializes Shape list, 386-387
Creating complex DirectDrawSurface, 136
Creating DirectSound buffer, 247
Creating DirectSound buffer from a WAV res, 251-252
Creating DirectSound object, 245
Creating node object, 342-374
creating offscreen DIB, 68-69
Creating offscreen DirectDrawSurface, 140-142
Creating offscreen surface, 187-188
Creating simultaneous sound buffers, 255-256
Creating surfaces for a window, 579
Creating primary surface, 136-137
Creating window for Windows 95, 14-15
Critical game code in the updateFrame() function, 367
cShapeList and cShapeListNode class declarations, 382-383
cShapeList member function RenderList(), 392
cTree member function InordertreeWalk(), 361-374
cTree member function InsertTree(), 358
Data structures representing VR user orientation, 596-597
DDTOOLS function writeSurface(), 276
DDTOOLS.CPP utility DDError(), 127
Declaration for a C++ cList class, 340
Declaration of a C++ node class, 339
Defining CELL in the terrain map, 159-160
Defining game GUID from the header file, 484
Defining GameSessionsDefs structure, 484-516
Defining location on a terrain map, 159
Defining polygon spaceship, 315
Defining simple protocol messages, 503
Defining spaceship from the point array, 320
Defining terrain map, 161
Definition of point structure, 314
Deleting the item once it's found, 347-348
Describing offscreen DirectDrawSurface, 140-143
Describing offscreen DirectDrawSurface using DDSURFACEDESC, 140
Design for a C++ list class, 339
Determining asteroid velocities, 389
Determining if a buffer, 258-259
Determining if a DirectDrawSurface, 144-152
Dialog box to join an active DirectPlay, 498
Dialog controls for selecting the session, 479-480
Dialog to host or join a game session, 483
DirectDraw method GetAttachedSurface(), 138
DirectDraw object instantiation, 126
DirectDrawSurface, 450-451

DlgProcHost driver function GetGameHost(), 483
drawBackGround() function, 451-452
drawBubonic() function, 428-429
drawCells() function, 445-447
Drawing a filled polygon, 590-591
Drawing Bubonic in any of his motion states, 194-219
Drawing cells Bresenham algorithm, 550-553
Drawing the array to the back surface, 591-592
Drawing the DIDRVR display, 198
Drawing the same cell for n milliseconds, 196
Drawing the Shape objects for Asteroids, 392
Drawing the SpaceFoil explosion, 332
Drawing the terrain tiles, 191
Drawing tiles from lpDDSOne to lpDDSBack, 165-166
drawItems() function, 457
drawTerrain() function, 422-423
Dungeon DirectPlay messaging structure, 544
Dungeon terrain map, 522-523
Emulating a display mode in window, 574
encodeSpriteRect() Shape class member function, 397
Encoding the logic arrays, 305
Enumerated list of game shape, 323
Example of a list object for a mechanized robot battalion, 352-353
Examples of page scripts used in the Xplore game, 264-265
Four-byte chunk ID declarations, 230
Freeing the Spells of Fury LPWAVINFO buffer, 256
Functions used for selecting the session, 480-481
Game host server, 510
Game process scheduler, 365
Generic Game Application's message loop, 149
generic game application's WinFunc, 23
Generic game application's WinMain(), 22-23
Get the i-glasses Tracker angles, 599
getJoystick() function, 214-215
Getting mouse updates on demand, 208-209
GetWAVRes() function, 236-237
Handling a collision with shape of type BIG_ROCK_INDEX, 411
Handling the mouse WM_LBUTTONDOWN message, 205
Handling the WM_KEYDOWN event for VK_NUMPAD6, 201
IDirectDraw COM interface, 120-121
IFF BitMapHeader structure, 230-231
Implementing the best fit allocation scheme, 582-583
InitDIBObjects transparency blitting, 80-81
Initializing DirectDraw in DIDRVR, 186
Initializing a DirectDrawSurface, 139
Initializing a level in Asteroids, 386
Initializing a Shape object, 325
Initializing a shape object array, 324
Initializing afterburner Shape objects, 325
initializing base tiles, 437
initializing DIDRVR application, 182-183
Initializing DirectDraw for 640x480, 131
Initializing DirectDraw for Mode X, 576-577
Initializing DirectDrawClipper object, 580
Initializing DirectInput, 212
Initializing DPSESSIONDESC to connect to session, 496
Initializing game shape objects, 385
Initializing DirectPlay game server, 487
Initializing Dungeon PLAYER linked list, 530-531
Initializing encoding logic arrays, 305
Initializing EnumSessions list box, 491
Initializing gameObj shape array, 385
Initializing GameSessionDef structure for game server, 486
Initializing generic game application wind, 21
Initializing PLAYER attributes, 538
Initializing PLAYER cell positions, 537
Initializing PLAYER fields, 535-536
Initializing session provider dialog, 478
Initializing spaceship Shape objects, 324
Initializing terrain map, 162
Initializing values in GameSessionDefs structure, 484
Initializing velocity vectors of space debris, 407
Initializing virtual world, 597
initSlices() function, 439-440
Inserting element into the tree, 359
Inserting into the cList object, 341-374
Inserting PLAYER into an empty list, 531
Inserting PLAYER into nonempty list, 533
ITEM linked list declarations, 454-455
ItemCalls lookup table, 455
ITEMS lists, initializing, 454-455
IUnknown COM interface, 121
Joining active game session, 489-491

Launching thread using the scheduler, 371-372
Linear updates on xz plane, 597-598
loading animation cells for your application, 185
Loading the RIFF data into the LPWAVINFO buffer, 254-255
Locating an available DirectSound buffer, 257-258
Locating the tile on the offscreen surface, 527-528
Locking the back flipping surface, 277
Message handling function for dialog box, 32
Message loop for the DIDRVR application, 198
Message processing used in NetChess, 512-513
Mouse structure used by the DIDRVR, 204-219
Moving a Shape object on the screen, 329-330
Moving Bubonic with DirectInput, 217
Moving Player, 539-542
Moving vertices 16 pixels up the y axis, 317
Moving vertices six pixels to the right, 316
NetChess NETCHESSMSG structure, 512
Normalizing an integer value, 216
Notifier for whose move it is in the chess game, 511
Notifying remote nodes PLAYER move, 545
Notifying the server of player move completion, 514
Opening a DirectPlay session, 497
Pausing a Direct X game, 369
Picking up items map, 556-557
Pitch to calculate DirectDrawSurface, 568
Placing PLAYER on terrain map, 536-537
PLAYER structure used in Dungeon, 529-530
Playing DirectSound using Spells of Fury, 257
processGame() function, 392, 395-396
processGame() function using encodeSpriteRect(), 398
processGame() used in the COLLIDE example function, 411-412
Processing Our Game Messages, 506
Processing Shape objects, 393-394
Processing the mouse updates from handleMouse, 206
Processing the WM_MOUSEMOVE message, 205
Querying your hardware for display modes, 575-576
Random number generator from Roids, 387
Reading a DirectPlay message, 505-506
Reading Messages from the Message Loop, 508
Reading PLAYER move update message, 546
Reading player turn message, 513
Reading the DUNGEON.INF file, 525-526
Reading the HELLOMSG, 507
Reading the player move completion message, 515
Recursive Inorder Traversal of a tree, 360
Rejecting a point code, 307
Removing the first item in the cList object, 345
Removing the item from the list, 351
renderSpriteRect() Shape class member function, 398
Requesting a list of game messages, 495-496
Restoring lost DirectSound buffers, 260
Retrieving and normalizing the joystick coordinates, 216
RIFF chunk structure CK, 231-232
rotateObjectLeft() function, 326
rotateObjectRight() function, 327
rotateSpaceFoilLeft() function, 326
Rotating a vertex, 318-319
Rotating an object with objectTransformation(), 327
Searching and removing items from the cList, 345
Searching for an item in the list, 347
Selecting a potion using the mouse pointer, 557-558
Selecting a session provider, 478-479
Sending the HELLOMSG using DirectPlay, 503
sequenceExecute() function, 267
Server HELLO message with a client ACK response, 504
SetClipBounds() function, 304
SetCooperativeLevel() for full-screen Mode X, 575
SetCooperativeLevel() for GC_WINDOWBOOL, 573
setDebrisVectors() function, 409
Setting a DirectDraw ColorKey value from a CDib, 156-157
Setting a display mode using DirectDraw, 130
Setting Mode X using a DOS4/GW 32-bit DOS extender, 112
Setting the cooperative level for DirectDraw, 129
Setting the DirectDraw palette from a DIB RGBQUAD palette, 154-155
Setting the DirectDrawPalette, 155
Setting the JOYINFOEX and JOYCAPS flags, 213
Setting the scanline entries in the array, 589-590
Setting the step value, 289

Setting the video to Mode 13h using DOS4/GW, 109
Setting up the DirectDraw palette, 187
sfArray point list, 322-323
Shape class definition, 319
Shape member function moveShape(), 330
Shape member function renderShape(), 331
Shape member function rotateShape(), 320
Simple DirectPlay message structure, HELLOMSG, 502
Simple Spells of Fury DSTOOLS driver, 261
Simple Windows 95 hello world application, 9
SLICE data structure, 434
slope, constructing tables, 427-428
slope equations, 425
Space debris vector recoil constants, 406
Spells of Fury callback function EnumGames, 492
Spells of Fury CreateWAVInfo() function, 253-254
Spells of Fury DDTOOLS and CDib Object, 157
Spells of Fury DDTOOLS routine, 141-142, 150-156
Spells of Fury DDTOOLS routine flipSurfaces(), 147-148
Spells of Fury DPTOOLS function SessionDriver(), 499-500
Spells of Fury DSError() utility, 246
Spells of Fury game console-style descriptor, 571
Spells of Fury Sound WAVINFO structure, 252
STEP structure, 266-267
Stop Bubonic using joystick button one, 218
The structure JOYDATA used for DirectInput, 211-219
Structure for the IFF chunk header, 229
Structure for the IFF form chunk, 228-229
Swapping y terms, 288
Task state TS_SPLASH, 366-367
terrain maps, looping, 442
TERRAIN.INF file, reading SLICE data structures, 435
Thread InitJoyThread(), 373
Tile CELL database, 524-525
Tile definitions used in PowerSuit, 160
tiles
    drawing, 443
    drawing to back flipping surfaces, 423
    lookup tables, 426-427
Tree class in C++, 356-374
Tree node class in C++, 356-374
Trivially accepting a point code, 306
Tumbling the space debris, 395
Unpacking the form chunk, 239-240
Unpacking the four-byte 'data' chunk, 243-244
Unpacking the four-byte 'fmt ' chunk, 243
Unpacking the four-byte 'RIFF' form chunk, 241-242
UnpackWAVChunk()function, 237-239
Update the y coordinate, 291
updateDebrisCollision() function, 403-404
updateFrame( ) function, 424
Updating PLAYER's position, 542
Updating the errorTerm and y coordinate, 295
Updating the game console, 583-584
Updating the new space debris objects for list insertion, 410-413
Updating the SpaceFoil velocity components, 328-329
Updating the x coordinate, 295
Updating using the Tracker, 600
Updating yaw displacements, 598
Using checkKeyboard() from within UpdateFrame, 203
Using DDTOOLS with the Generic Game Application, 131-132
Using DOS4/GW to perform a write to a video plane, 114
Using DSTOOLS in the children's Xplore game, 263
Using GetAsyncKeyState() to capture mouse button input, 209
Using handleMouse() from updateFrame(), 207-219
Using lines to determine visibility, 549-550
Using potions in Dungeon, 558-559
Using the Generic Game Application and CDib to display a .BMP file, 60-62
Using the void* pointer to reference an arbitrary data type, 352
Using Windows messaging to capture keyboard input, 199-200
Walking the terrain map, 164-165
Windows 95 Console window driver for the cList class, 354
Windows 95 console window driver for the cTree class, 362

Windows friendly message loop, 364
WndProc, message handlingLRESULT APIENTRY, 17
Writing to a DirectSound buffer, 248-249
Writing to a streaming buffer using DirectSound, 249-250
Writing WAV data to a DirectSound buffer, 250-251
lists
ITEMS, 452-453
collision detections, 457-459
drawing, 456-457
initializing, 453-456
iteration, 390-393
linked, 337-354
C declaration, 338
defined, 337
objects, 338-355
loading animation cells, 183-186
Loading the animation cells for your application (listing), 185
Loading the RIFF data into the LPWAVINFO buffer (listing), 254
LoadResource() function, 236
local area networks, *see* LANs
locating tiles, 443-444
Locating an available DirectSound buffer (listing), 257
Locating the tile on the offscreen surface (listing), 527-528
Lock() function, 249, 275
locking surfaces, 277
Locking the back flipping surface (listing), 277
lookup tables, 426-427
ItemCalls, 455
slope, 427-428
tiles, 421-422
loops
images, copying, 73
message loops, 27-28, 198
surfaces, locking, 277
terrain maps, 442
y coordinates, drawing lines, 295-299
LOWORD() macro, 26, 205
lpbinfo data member, typecasting, 47
lpDDPal pointer, 187
lpDDSOffscreen pointer, 273
LPRGBQUAD data member, 49
lread() function, 58

M

MAC (Media Access Control), addresses, 471
macros
CW_USEDEFAULT, 13
HIWORD, 205
HIWORD(), 26
LOWORD, 205
LOWORD(), 26
SOCK_STREAM, 474
SW_MAXIMIZED, 14
SW_MINIMIZED, 14
WM_KEYDOWN, 25
WM_MOUSEMOVE, 25
WM_PAINT, 25
main() function, 8
Majority 12, 81-99
MakeSoundBuffer() function, 251
Manhole children's game, 261-267
mapping
addresses, 469-470
memory, DirectDrawSurface, 273
objects, offscreen, 317
maps
games, 300-302
terrain maps, 420-421, 521-528
drawing slices, 440-441
horizontal scrolling terrain maps, 521-528
initializing, 524-526
locating tiles, 526-528
looping, 442
non-linear, 424
player movements, 538-547
scrolling, 430
slicing, 432
vertical scrolling terrain maps, 521-528
zeros, 420
tiles, 418-419
maximizing applications, 368-369
MDK (Microsoft Multimedia Development Kit), 227
member functions
CDib class, 43-45
CheckCollision(), 400-401, 403
CreatePlayer(), 501
encodeSpriteRect(), 397
EnumSessions(), 492
InorderTreeWalk(), 357, 361-374
InsertList(), 341
InsertTree(), 357-358

Open(), 487, 497-498
Receive(), 506
RemoveList(), 412
RenderList(), 392
renderShape(), 392
renderSpriteRect(), 398
Send(), 503
SetPalette(), 187
memory
13h video mode, 109
allocation, verifying, 46
bank-switched memory, 568
BITMAPINFO structure, allocating, 46
buffers, DIBs, 41, 45
DIBs, 41
DirectDrawSurface, coordinates, 273
display coordinates, DirectDrawSurface translations, 273-274
image bits
allocating, 50
reading, 59
locked, accessing, 275
planes, 48
video, DOS extenders, 107
video drivers, configuration, 109
video mode X, 110-114
memory architecures
protected mode, 178-181
real mode, 174-178
DOS functions, 176-177
interrupt vector tables, 175-176
interrupts, 174-175
keyboard input, 177-178
memset() function, 50
Message handling function for the dialog box (listing), 32
Message loop for the DIDRVR application (listing), 198
message loops, 15-16, 198
Message processing used in NetChess (listing), 512
message processors, DirectPlay, 505-506
messages
DirectPlay
processing, 506
reading, 505-506, 508
sending, 503
structure, 502
LB_GETITEMDATA, 496
LBN_DBLCLK, 480
NetChess, 512
WM_COMMAND, 479
WM_INITDIALOG, 477, 492, 495
WM_KEYDOWN, PowerSuits, 86
WM_MOUSEMOVE, 204
WM_PAINT, 15
messaging
game applications, 25
game message loops, 27-28
handling, 28-29
Win32 messaging, 24-30
Windows 95, 25-27
Microsoft Developer Studio, dialog editor, 482
minimizing
applications, 367-368
collision detection, 402-404
lag time, 371-372
Mode X
DirectDraw
flipping surfaces, 577-584
initializing, 571-577
surface creation, 578
video cards, 567-569
display settings, 574-577
flipping surfaces, creating, 581-583
full-screen cooperative level, 574-577
video Mode X, 566-570
Mode X: The Medium of the Mystics, 566-584
modes, video
13h, 108-110
memory, 113
page-flipping, 110-114
mouse
button input, capturing, 209
input, 204-210
input messages, WM_LBUTTONDOWN, 205
messages, processing, 204-205
messaging, bypassing, 208-210
movements, tracking, 204-207
Mouse structure used by the DIDRVR application (listing), 204
movements, players, distributed environments, 544-546
moveObject() function, 329
moveShape() function, 329
MoveSuit() function, 86
Moving a Shape object on the screen (listing), 329-330
Moving Bubonic with DirectInput (listing), 217
Moving Player (listing), 539-542

Moving vertices 16 pixels up the y axis (listing), 317
Moving vertices six pixels to the right (listing), 316
MS-DOS, 173-174
  bypassing, INT 16h (interrupt 16h), 177-178
  protected mode, 178-181
  real mode, 174-178
    functions, 176-177
    interrupt vector tables, 175-176
    interrupts, 174-175
    keyboard input, 177-178
multimedia, 593
multiple collisions, 411-412
multitasking
  nonpreemptive, 363
  preemptive, 364-365
  Windows 95, 179-219, 363-364

N

NCMontyHall() function, 515
NCMoveCompleteNotify() function, 514
NCReadMove() function, 514-515
NCReadTurn() function, 513
NCTurnNotify() function, 511
NetChess, 508-510
  clients, 513-514
  messages
    processing, 512
    reading, 513-514
    structure, 511-512
  player turns, determining, 510-516
NetChess NETCHESSMSG structure (listing), 512
NetMinder, 486
NetWare, 471-472
Network General sniffers, 486
network layer, 468
networks, 465-467
  blocking, 475
  Ethernet, 465-467, 469
  ports, 472-473
  protocols, 467-472
    application layer, 467
    ARP, 469-470
    IPX (Internet Packet eXchange), 471-472
    link layer, 469
    network layer, 468
    transport layer, 468
    UDP (User Datagram Protocol), 470-471
  sniffers, 473, 486
  sockets, 473-476
new operator, 341
newItem( ) function, 455
node objects, 337
  classes, declaring, 339
  creating, 342
  finding, 343-345
  inserting, 341
  removing, 345, 347-348, 351
nodes, 482-483
non-linear terrain maps, 424
nonconvex polygons, 313-314
nonpreemptive multitasking, 363-364
Normalizing an integer value (listing), 216
notations, linked lists, 337
Notifier for whose move it is in the chess game (listing), 511
Notifying remote nodes PLAYER move (listing), 545
Notifying the server of player move completion (listing), 514
NULL RECT pointer, 275
NumberOfColorEntries() function, 58
numbers, random number generators, 387

O

object-oriented lists, *see* linked lists
object-oriented programming, *see* OOP
objects
  *pDib, 185
  animating, updating, 329-330
  animating, 326-327
    polygons, 316
    speed, 328-330
  CDib
    animation, 81-99
    initializing, 66, 183-219
  DIBs
    creating, 45-53
    creating offscreen, 68-69
  DirectDraw
    creating, 125-127, 186
    initializing, 124-132
  DirectDrawPalette, 49
  DirectPlay
    creating, 476
    initializing, 486-489
  DirectSound, creating, 244-252
  head, declaring, 340-341
  initializing, 324
  list, creating, 338-355
  mapping, offscreen, 317

moving, 448-452
nodes
creating, 342
inserting, 341-343
removing, 343-351
pDibBckGrnd, 66
pDibOffscreen, 66
Shape class, creating, 320-325
Shape objects, 381
collisions, 396-405
defining, 383
rotating, 395
shapes, declaring arrays, 322
tail, declaring, 340-341
updating, rendering, 330-331
objectTransformation() function, 327
offscreen surfaces,
creating, 187-188
DirectDraw, 139-152
OOP (object-oriented programming), 42
Open() member function, 487, 497-498
opening
DirectPlay sessions, 497
TERRAIN.INF file, 432-433
windows, virtual worlds, 162-166
Opening a DirectPlay session (listing), 497
operating systems
MS-DOS, 173-174
Windows 95, 179-180
operators
delete, 348
GlobalAllocPtr(), 45
GlobalFreePtr(), 45
new, 341
scope, CDib::, 44
Win32, 45
orientation, lines, checking, 285-286

P

P key, 369
packet switching, 465
packets, 465
page-flipping
flicker, 114-116
video modes, 110-114
palettes
DirectDraw, 152-157
color keys, 155-166
creating, 187-219
setting, 152-155
initializing, 49
passive sockets, 474
pausing applications, 369
Pausing a Direct X game (listing), 369
PCMWAVEFORMAT files, 235
PCX files, 39
*pDib object, 185
pDibBckGrnd object, 66
pDibOffscreen object, 66
PeekMessage() function, 28, 363
physical addresses, 174
Picking up items map (listing), 556-557
pickUpItem() function, 556
pitch, video cards, 568
Pitch to calculate DirectDrawSurface (listing), 568
pixels
DirectDrawSurface, writing, 276
display coordinates, DirectDrawSurface translations, 273-274
step values, 289
video frame buffers, writing, 272
pJoyData structure, initializing, 212
PL 0 (protection layer 0), 179
Placing PLAYER on terrain map (listing), 536-537
PLAYER linked list (Dungeon), 529-533
PLAYER node, initializing, 533-537
PLAYER structure used in Dungeon (listing), 529-530
player turns, determining, 510-515
playerMove() function, 541
players
attacking, 553-555
attributes, 537-538, 547-559
Dungeon, 528-538
fields, view, 547-553
movements
distributed environments, 544-546
terrain maps, 538-547
positions, updating, 539-544
RPGs
designing for multiple palyers, 528-546
health, 553-559
Playing DirectSound using Spells of Fury PlayWAVInfo() (listing), 257
plotting pixels, 272
POINT (pt) variable, 209
Point-Slope equation, 281-282
pointers
BITMAPINFOHEADER, assigning values, 47
COM interfaces, 119

DirectDrawSurface, accessing buffers, 275
image bits, 64, 72
initializing, 346
locked surfaces, assigning, 277
lpDDPal, 187
lpDDSOffscreen, 273
NULL RECT, 275
pvRes, 383
void*, 352
polygons
animating, 316
asteroids, definitions, 383
classes, creating, 319
collisions, detecting, 412
convex compared to nonconvex, 313-314
declaring arrays, 322
defining, 314-316
distance translation, 316-317
filled polygons, 584-593
BSP (Binary Space Partitioning) trees, 592
convex polygon scan conversions, 585-593
directional vectors, 585
drawing, 590-592
IDs (polyIDs), 383
rotating, 318-319
SpaceFoil game, 321
polyIDs (polygon IDs), 383
ports, 472-476
positions, players, updating, 539-544
PostQuitMessage() function, 370
posx field, 189, 201
posy field (wartHog struct), 189
potential collisions, detecting, 400
potions, 555-557
PowerSuits, 81-82, 158-166
movement, coding, 82-90
parameters, 86
rendering, 90
WM_KEYDOWN messages, 86
primary surfaces
DirectDraw, 134-139
windows, 578-580
processGame() function, 390-396, 411-413
processGame() used in the COLLIDE example function (listing), 411-412
processing
input, joysticks, 214-215
messages
DirectPlay, 506
mouse, 204-205
NetChess, 512-513
objects, list objects, 390-393
Processing Our Game Messages (listing), 506
Processing Shape objects (listing), 393-394
Processing the mouse updates from handleMouse() (listing), 206
Processing the WM_MOUSEMOVE message (listing), 205
processors
8086, 174
protected mode addressing, 106
programming
Super PowerSuits, tiles, 158-162
trends, 315-316
Windows 95, 19
protected mode, 178-181
addressing, 106
DOS, 105
interrupts, 178
protection layer 0 (PL 0), 179
protocols, 467-472
ARP (Address Resolution Protocol), 469-470
IPX (Internet Packet eXchange), 471-472
layers
application, 467
link, 469
network, 468
transport, 468
messages, defining, 502-503
UDP (User Datagram Protocol), 470-471
pt (POINT) variable, 209
pvRes pointer, 383
pwrSuitInfo structure, 86
PwrSuitInfoStruct structure, 86
pwrSuitPosX field, 86
pwrSuitPosY field, 86

Q-R

Querying your hardware for display modes (listing), 575-576
QueryInterface() function, 121

rand() function, 387
randNumGen() function, 387
random disorder, creating, 393-396
Random number generator from Roids (listing), 387
random number generators, 387
ReadFile() function, 54
ReadHelloMsg() function, 507
reading
messages, 505-508

RIFF files, 235-244
tiles, SLICE databases, 435
Reading a DirectPlay message (listing), 505
Reading Messages from the Message Loop (listing), 508
Reading PLAYER move update message (listing), 546
Reading player turn message (listing), 513
Reading the DUNGEON.INF file (listing), 525-526
Reading the HELLOMSG (listing), 507
Reading the player move completion message (listing), 515
readTracker() function, 599
real mode addressing, 174-178
DOS functions, 176-177
interrupt vector tables, 175-176
interrupts, 174-175
keyboard input, 177-178
real mode environments, protected modes, compared, 106
RealizePalette() function, 117
Receive() function, 476, 506
receiving messages, DirectPlay, 505-508
recoil constants, determining, 406
RECT data structures, 190
images, locations, 70-75
sprites, calculating grid coordinates, 90-91
rectangles, bounding, 396-402
recursion, 357-363
Inorder Traversal algorithm, 360-362
insertions, 357-359
Recursive Inorder Traversal of a tree (listing), 360
recv() function, 475
referencing frames, virtual reality, 594-598
region codes, line-clipping, calculating intersections, 310
RegisterClass() function, 11
reject() function, line-clipping, 307
Rejecting a point code (listing), 307
releasePiece() function, 514
reliable transmissions, 470
RemoveList() member function, 412
removing
node objects, 343-348, 351
Shape objects, 411
Removing the first item in the cList object (listing), 345
Removing the item from the list (listing), 351
rendering
objects, updating, 330-331
PowerSuits, 90
sprites, 69-81
tiles, 422-424, 450-452
see also drawing
RenderList() member function, 392
renderShape() member function, 392
renderShapeList() function, 399-400
renderSpriteRect() member function, 398
renderSpriteRect() Shape class member function (listing), 398
REQs (requests) (ARP), 494
requesting game messages, 495-496
Requesting a list of game messages (listing), 495
requestTracker() function, 599
RESs (responses), ARP, 494
Restore() function, 144
restoring surfaces, DirectDraw, 143-144
Restoring lost DirectSound buffers (listing), 260
retrieving tiles, 443-444
Retrieving and normalizing the joystick coordinate (listing), 216
RFCs (Requests For Comments), 470
RGBQUAD color table, 40-42
copying, 187
creating, 49
DIB objects, 45
reading, 58-59
RIFF (Resource Interchange File Format), 227-233
chunks, 231-232, 237-242
forms, 232-233
reading, 235-244
RIFF chunk structure CK (listing), 231
ring protection, layers, 105
RLE (run-length encoding), 48
rotateObjectLeft() function, 326, 393
rotateObjectRight() function, 393
rotateShape() function, 320
rotateSpaceFoilLeft() function, 326
rotating
polygons, 318-319
Shape objects, 395
Rotating a vertex (listing), 318-319
Rotating an object with objectTransformation() (listing), 327
routers, 465
RPGs (role-playing games), 519
multiple players, designing for, 528-546
players, 528-538, 553-559
potions, 555-557
run-length encoding, *see RLE*

S

scheduler, 390-413
scheduling game events, 364-371
  game process schedulers, 365
  TS_CRITICAL, 367
  TS_SPLASH, 366-367
  TS_SUSPEND, 369
  TS_TERMINATE, 370
scope operators, CDib class, 44-45
screen resolutions, games, 566
screens, updating, 62
scrolling terrain maps, 430
searching linked lists, 347
Searching and removing items from the cList object (listing), 345
Searching for an item in the list (listing), 347
segment registers, 174
selecting session providers (DirectPlay), 478-479
Selecting a potion using the mouse pointer (listing), 557-558
Selecting a session provider (listing), 478
Send() function, 476, 545
Send() member function, 503
sending messages, DirectPlay, 503-504
Sending the HELLOMSG using DirectPlay (listing), 503
sequenceExecute() function (listing), 266
Server HELLO message with a client ACK response (listing), 504
servers
  DirectPlay, 485-488
  sockets, 474
session providers (DirectPlay), 477-478
SessionDriver() function, 499-500
sessions
  DirectPlay
    client/server communication, 502
    initializing, 476
    joining, 489-491
    opening, 497
    session providers, 477-481
  WinSock, 494
sessions providers, 478-480
SetCDibState() function, 56
SetClipBounds() function (listing), 304
SetCooperativeLevel() for full-screen Mode X (listing), 575
SetCooperativeLevel() for GC_WINDOWBOOL (listing), 573
SetCooperativeLevel() function, 127, 129
SetCursorPos() function, 206
setDebrisVectors() function, 406, 408-409
setDisplayMode() function, 119, 129
SetFocus() function, 22
SetPalette() function, 153, 187
SetRGBQuadPalette() function, 187
setScanLineEntry() function, 590
setting palettes, DirectDraw, 152-155
Setting a DirectDraw ColorKey value from a CDib object pixel (listing), 156
Setting a display mode using DirectDraw (listing), 130
Setting Mode X using a DOS4/GW 32-bit DOS extender (listing), 112
Setting the cooperative level for DirectDraw (listing), 129
Setting the DirectDraw palette from a DIB RGBQUAD palette (listing), 154
Setting the DirectDrawPalette from the CDib object (listing), 155
Setting the JOYINFOEX and JOYCAPS flags (listing), 213
Setting the scanline entries in the array (listing), 589-590
Setting the step value (listing), 289
Setting the video to Mode 13h using DOS4/GW (listing), 109
Setting up the DirectDraw palette (listing), 187
SetTransColorKey() function, 188
SetTransparentColor() function, 157
sfArray point list (listing), 322-323
sfCheckCollision() function, 403, 412
Shape class
  creating, 319
  member functions
    CheckCollision(), 400-401, 403
    encodeSpriteRect(), 397
    renderSpriteRect(), 398
  SpaceFoil game, creating objects, 321-325
  spaceship objects, creating, 320-321
Shape class definition (listing), 319
Shape member function moveShape() (listing), 330
Shape member function renderShape() (listing), 331
Shape member function rotateShape() (listing), 320
Shape objects, 381
  arrays, gameObj, 384
  bounding rectangles, assigning, 396-402

collisions, 396
definitions, 383-390
drawing, 392
random movements, creating, 393-394
rotating, 395
tumbling, 393-396
variables, initializing, 385
Shape::encodeObject() function, 325
Shape::setAngle() function, 325
shapes, code, modifying, 323
ShowCursor() function, 22
ShowWindow() function, 14, 368
Simple DirectPlay message structure, HELLOMSG (listing), 502
Simple Mail Transfer Protocol (SMTP), 467
Simple Network Management Protocol (SNMP), 467
Simple Spells of Fury DSTOOLS driver (listing), 261
Simple Windows 95 "hello world" application (listing), 9
SKDs, 118
SLICE data structure, 434-435
slices, drawing, 440-443
slicing
columns, tiles, 431-440
terrain maps, 432
sliding windows, slices, drawing, 441-442
slope
Bubonic the WartHog, drawing, 428-430
equations, 425-427
lines, drawing, 279-281
lookup tables, 427-428
SMTP (Simple Mail Transfer Protocol), 467
sniffers (networks), 473, 486
SNMP (Simple Network Management Protocol), 467
SOCK_STREAM macro, 474
socket() function, 474
sockets, 473-474
descriptors, 474
WinSock API (Windows Socket application programming interface), 475
sound data structures, 252-267
source files, DIGENX.CPP, 207
Space debris vector recoil constants (listing), 406
SpaceFoil game, 321-325
SpaceHulk, Electronic Arts, 224
Spells of Fury callback function EnumGames() (listing), 492
Spells of Fury CreateWAVInfo() function (listing), 253
Spells of Fury DDTOOLS routine (listing), 141
Spells of Fury DDTOOLS routine flipSurfaces() (listing), 147
Spells of Fury DDTOOLS routine to blit surfaces (listing), 151
Spells of Fury DDTOOLS routine to erase a DirectDraw surface (listing), 149
Spells of Fury DDTOOLS routine to set a DirectDraw ColorKey (listing), 155
Spells of Fury DPTOOLS function (listing), 499
Spells of Fury DSError() utility (listing), 246
Spells of Fury game console-style descriptor (listing), 571
Spells of Fury Sound WAVINFO structure (listing), 252
Sphinx, 417-418
sprites
animation, updating, 189
bounding rectangles, 396-402
Bubonic the WartHog
cell arrays, 192-195
displaying, 198
HOG_GO_RIGHT, 197
HOG_STOP_LEFT, 196-197
HOG_STOP_RIGHT, 195-196
cell arrays, animating, 192-199
collisions
detecting, 396-404
elastic collisions, 404-405
recoil, 406
color, transparency, 75-78
DIB objects, copying, 70
drawing, DCs, 80-81
flicker, 62
grid coordinates, calculating, 90-91
polygons, rotating, 320
rendering, 69-81
srcIndex value, 76
standLeftRect array, 197
standRightRect array, 196
states, reading, 54-60
STDIO.H header file, 9
STEP structure used to describe a series of events (listing), 265
stereoscopic headsets, 593-601
Stop Bubonic using joystick button one (listing), 218

Stopping Bubonic with facing HOG_STOP_RIGHT (listing), 196
StretchDIBits() function, 51
Structure for the IFF chunk header (listing), 229
Structure for the IFF form chunk (listing), 228
structure JOYDATA used for DirectInput (listing), 211
structures
    CELL, tiles, 421
    DDSURFACEDESC, 275
Super PowerSuits, 158-166
SuperNova, Big Five software, 223
surfaces
    2D surfaces, 3D composition, 585
    DirectDraw, 133-152
        blitting, 145-152
        creating (Mode X), 578
        flipping, 145-152
        flipping (Mode X), 577-584
        initializing, 135
        offscreen surface, 140-152
        primary surface, 134-139
        restoring, 143-144
    flipping surfaces, Mode X, 581-583
    locking, 277
    offscreen, creating, 187-188
    primary surfaces, windows, 578-580
SW_MAXIMIZED macro, 14
SW_MINIMIZED macro, 14
swapping endpoints, 288
Swapping y terms (listing), 288
swapTermsInt() function, 288
switch statement, 486
system calls, 473-474
system requirements, trends, 315-316

T

tables
    IDTs (interrupt descriptor tables), 178
    interrupt vector tables, 175
    lookup, 426-427
        slope, 427-428
        tiles, 421-422
    RGBQUAD color table, copying, 187
tail objects, declaring, 340-341
Task state TS_SPLASH (listing), 366
taskState variables, 365
    TS_CRITICAL, 367
    TS_NEW_THREAD, 371
    TS_SPLASH, 366-367
    TS_SUSPEND, 369
    TS_TERMINATE, 370
TCP (Transmission Control Protocol), 468
TCP/IP (Transport Control Protocol/Internet Protocol), 467, 472
telnet, 467
terminating applications, 369-370
terrain maps, 162, 420-421, 521-528
    Bubonic the WartHog listing, 433
    horizontal scrolling terrain maps, 521-528
    initializing, 524-526
    looping, 442
    non-linear, 424
    players, movements, 538-547
    scrolling, 430
    slices, drawing, 440-441
    slicing, 432
    tiles
        locating, 526-528
        lookup tables, 421-422
    vertical scrolling terrain maps, 521-528
    zeros, 420
TERRAIN program, 447-452
terrain tiles, drawing, 191-199
TERRAIN.INF file
    initSlices( ) function, 439-440
    opening, 432-433
    SLICE data structure, reading, 435
TerrainMap arrays, 420-421
TERRIAN.INF file, SLICE databases, reading tiles, 435
theta angles, calculating, 406
Thread InitJoyThread() (listing), 373
threads, 364
    concurrent, 371-374
    creating, 371-372
thruster() function, 328
tickDiff value, 196
tilde (~), 43
Tile CELL database (listing), 524-525
Tile definitions used in PowerSuit (listing), 160
tiles, 418-419
    animations, 444-446
    Bubonic the WartHog, 430
        drawing with slope, 428-430
        tile CELL structure, 421
    CELL structures, 421
    columns, slicing, 431-440
    drawing, 423, 443
    initializing, 436-440
    locating, 443-444
    lookup tables, 421-422, 426-428

maps, 419-421
moving objects, 448
reading SLICE databases, 435
rendering, 422-424, 450-452
slope, endlevel values, 426-427
slope equations, 425-427
Super PowerSuits, programming with, 158-162
terrain maps
locating, 526-528
non-linear, 424
scrolling, 430
TERRAIN.INF file, opening, 432-433
token passing, NetChess, 508-515
tournament games
NetChess, 508-516
player turns, determining, 510-515
tracking
head movements, headsets, 594
mouse movements, 204-207
TranslateMessage() function, 16, 29
Transmission Control Protocol (TCP), 468
transparent blit operations, 78
Transport Control Protocol/Internet Protocol, *see TCP/IP*
transport layer, 468
Tree class in C++ (listing), 356
Tree node class in C++ (listing), 356
trees, binary, 354-362
trivialAccept() function, 306-307
Trivially accepting a point code (listing), 306
Tron, 594
TS_CRITICAL state, 367
TS_NEW_THREAD, 371
TS_SPLASH state, 366-367
TS_SUSPEND, 369
TS_TERMINATE, 370
tumbleDebris() function, 395
tumbling Shape objects, 393-396
Tumbling the space debris (listing), 395
two-player games, RPGs, designing, 528-546
typecasting data members, lpbinfo, 47
TYPEDEFed variables, 11

U

Uccello, Paolo, 584
UDP (User Datagram Protocol), 468, 470-471
datagrams, 471, 494
IPX (Internet Packet eXchange), 471-472
UNIX, BSD UNIX, blocking, 475-516
Unlock() function, 251, 275
unpacking
chunks, RIFF files, 237-242
data chunks, Waveform, 243-246
format chunks, 242-243
Unpacking the form chunk (listing), 239
Unpacking the four-byte 'data' chunk (listing), 243
Unpacking the four-byte 'fmt ' chunk (listing), 243
Unpacking the four-byte 'RIFF' form chunk data (listing), 241
UnpackWAVChunk() function, 247
UnpackWAVChunk() function (listing), 237
Update the y coordinate (listing), 291
updateDebrisCollision() function, 403-409
updateFrame() function, 198, 207, 367, 424
UpdatePlayerLeft() function, 600
UpdatePlayerRight() function, 600
UpdateScreen() function, 66
UpdateVRUser() function, 597
UpdateWindow() function, 15
updating
DCs, screens, 66
game console, 583-584
objects, 329-331
players, positions, 539-544
screens, 62
sprite animation, 189
Updating PLAYER's position (listing), 542
Updating the errorTerm and y coordinate (listing), 295
Updating the game console (listing), 583-584
Updating the new space debris objects for list insertion (listing), 410-413
Updating the SpaceFoil velocity components (listing), 328-329
Updating the x coordinate (listing), 295
Updating using the Tracker (listing), 600
Updating yaw displacements (listing), 598
User Datagram Protocol (ARP), 470-471
User Datagram Protocol (UDP), 468
user interfaces, *see* interfaces
Using a WndProc for message handling (listing), 17
Using checkKeyboard() from within UpdateFrame() (listing), 203
Using DSTOOLS in the children's Xplore game (listing), 263
Using GetAsyncKeyState() to capture mouse button input (listing), 209
Using handleMouse() from updateFrame() (listing), 207
Using lines determine visibility (listing), 549-550

Using potions in Dungeon (listing), 558-559
Using the void* pointer to reference an arbitrary data type (listing), 352
Using Windows messaging to capture keyboard input (listing), 199

V

values
  BITMAPINFOHEADER, 48
  images, scanline offsets, 76
  pixels, step, 289
  srcIndex, 76
  tickDiff, 196
  transparency bytes, retrieving, 77
variables
  binfoSize, 46
  bIsActive, 365
  taskState, 365
  TYPEDEFed, 11
values, assigning to BITMAPINFOHEADER pointers, 47
vectors
  direction, determining, 406
  velocity, 388, 395
velocity vectors, 395
  determining, 388, 406-413
  initializing, 407
vertical scrolling terrain maps, 521-528
vertices, reading, 322
video
  architecture, DOS, 108
  drivers
    blitting, 107
    GDI, 117
    memory configuration, 109
  frame buffers, writing pixels, 272
  memory, DOS extenders, 107
  modes
    13h, 108-110
    memory, 113
    page-flipping, 110-114
    X, 110
video cards, memory architectures, 567-569
video Mode X, 566-570
video modes, DirectDraw, 129-134
view fields, players, 547-553
Virtual I/O i-glasses headsets, 593-601
virtual keys, 200, 370
virtual Mozart, 267
virtual reality, 593, 601
  filled polygons, 584-593
  frame referencing, 594-598
  headsets, head movement tracking, 594
  mechanics, 593-601
  stereoscopic headsets, 593-601
virtual worlds, windows, opening, 162-166
VK_ESC virtual key, 370
VK_NUMPAD4 key, 200
VK_NUMPAD5 key, 201-204
VK_NUMPAD6 key, 200-201
VKs (virtual keys), 200, 370
void* pointer, 352

W-Z

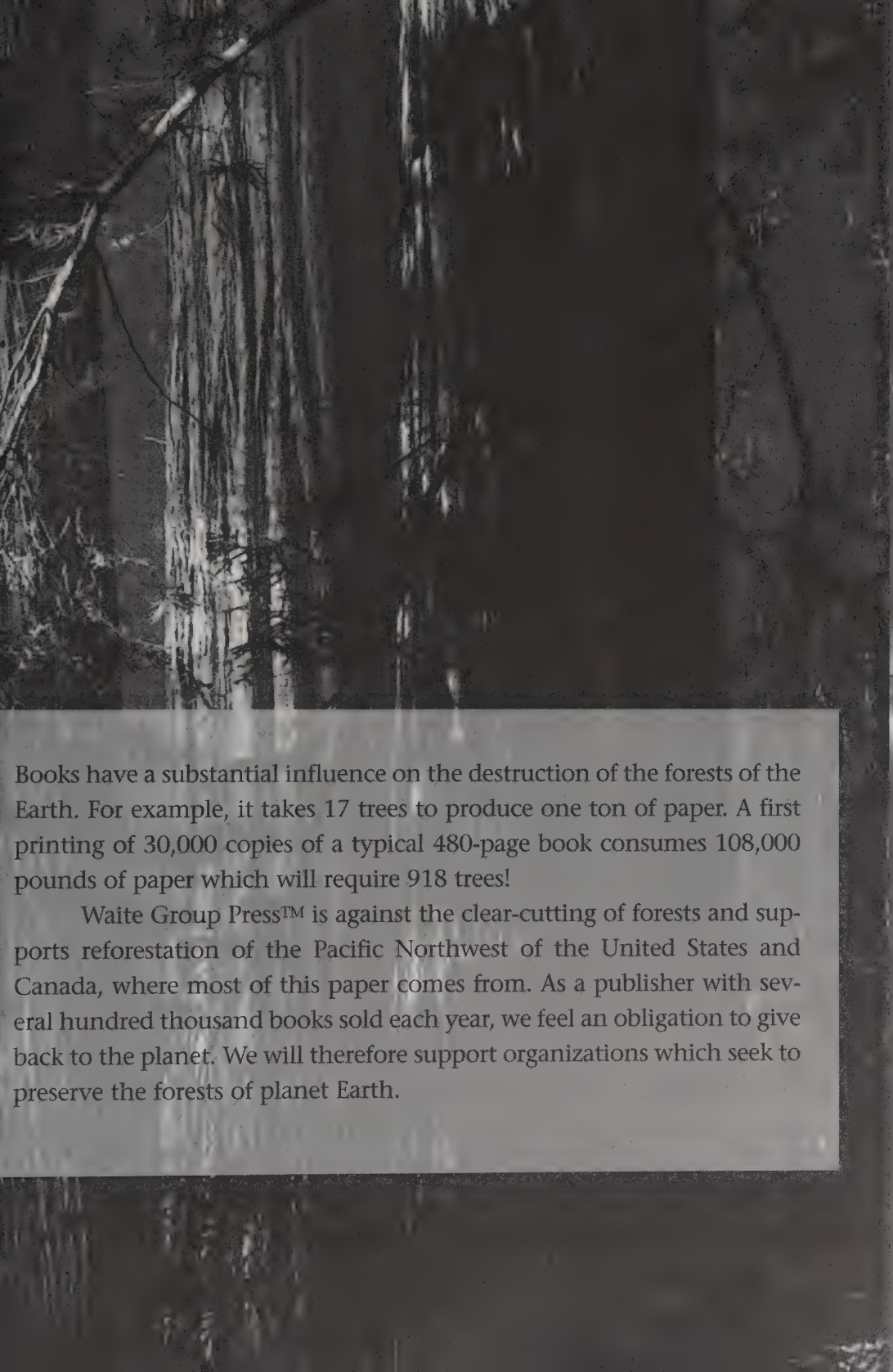
Walking the terrain map (listing), 164
WANs (wide area networks), 465
wartHog struct, 189
  case statements, 195-197
  fields, 189-190
Watcom C++ run-time library, 178
WAVE files, 235-237, 243-246
WAVEFORMAT data structure, 233-235
Waveforms, writing, DirectSound buffers, 247-252
WAVEINFO, 253-267
wide area networks, *see* WANs
Win32 API, 7
Win32 messaging, 24-30
Win32 operators, 45
Windows
  APIs, WinSock, 475
  DOS extenders, 106
  files, .BMP, 40
  GDI, 116-118
  graphics image formats, DIBs (Device-Independent Bitmaps), 37-99
  macros, 205
  messages
    LB_GETITEMDATA, 496
    WM_INITDIALOG, 477, 492, 495
  multitasking, 363
windows
  boundaries, 301-307
  creating, applications, 11-15
  dialog boxes, 30-33
  game console window, initializing, 20-22
  opening, virtual worlds, 162-166
  primary surfaces, 578-580
  sliding, drawing slices, 441-442

Windows 3.1, blocking, 475
Windows 95
  applications, writing, 8-19
  console applications, 362
  DirectPlay API, 476
  game consoles, creating, 19-30
  messaging, 179-180
  multitasking, 363-364
  programming, 19
  user input, 199-219
    keyboards, 199-204
    mouse, 204-210
Windows 95 Console window driver for the cList class (listing), 354
Windows 95 console window driver for the cTree class (listing), 362
Windows friendly message loop (listing), 364
Windows Socket API, 475
WINDOWS.H heaader file, 9
windows.h header file, 200
WinFunc() function, 67
WinG API, 107
WinMain() function, 8-11, 15, 198
  generic games, 22-23
  initPlayer() function, 531
WinSock, 475, 494
WinSock API (Windows Socket application programming interface), 475
winsock.dll, 475
WM_ACTIVATEAPP activation flag, 369
WM_COMMAND message, 479
WM_INITDIALOG message, 477, 492, 495
WM_KEYDOWN, 199-204
  VK_NUMPAD5 key down, 201-202
  VK_NUMPAD6 key down, 200-201
WM_KEYDOWN macro, 25
WM_KEYDOWN messages, PowerSuits, 86
WM_LBUTTONDOWN case statement, 205
WM_MOUSEMOVE macro, 25
WM_MOUSEMOVE message, 204
WM_PAINT macro, 25
WM_PAINT message, 15
WM_RBUTTONDOWN, 205
WNDCLASS, 11-13
WndFunc() function, 12, 16-19, 199-200, 204
world coordinate systems, game maps, 300-301
writeSurface() function, 276
Writing to a DirectSound buffer, 248-249
Writing to a streaming buffer using DirectSound (listing), 249
Writing WAV data to a DirectSound buffer (listing), 250
WSAAsyncGetHostByName() function, 475
WSAIsBlocking() function, 475

Xerox Network Standard (XNS), 471
Xerox Parc, Ethernet, 465
Xplore games, DSTOOLS, 262-263

y-intercept, lines, drawing, 281-282

zero, terrain maps, 420

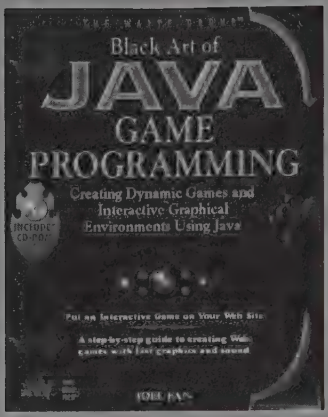


ENVIRONMENTAL AWARENESS

Books have a substantial influence on the destruction of the forests of the Earth. For example, it takes 17 trees to produce one ton of paper. A first printing of 30,000 copies of a typical 480-page book consumes 108,000 pounds of paper which will require 918 trees!

Waite Group Press™ is against the clear-cutting of forests and supports reforestation of the Pacific Northwest of the United States and Canada, where most of this paper comes from. As a publisher with several hundred thousand books sold each year, we feel an obligation to give back to the planet. We will therefore support organizations which seek to preserve the forests of planet Earth.

WAITE GROUP PRESS™

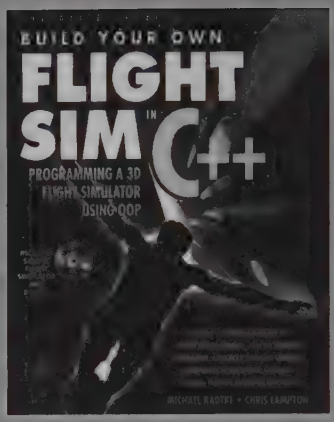


THE BLACK ART OF JAVA GAME PROGRAMMING

Joel Fan

Black Art of Java Game Programming covers all the essentials of Java 1.1 for graphics and game development. This comprehensive guide demonstrates how to write a cutting-edge interactive video game for the Web, playable by millions around the world on practically every platform. The unique networking techniques showcased in this valuable reference are useful for a wide range for multi-user environments in addition to game programming.

Available October, 900 pages
ISBN: 1-57169-043-3
U.S. $49.99
1 – CD-ROM



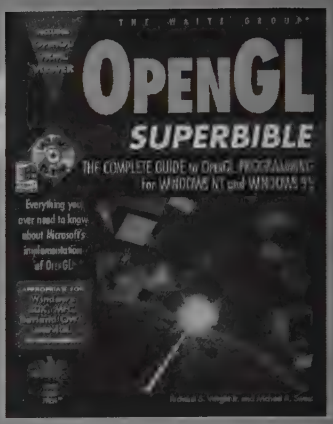
BUILD YOUR OWN FLIGHT SIM IN C++

Michael Radtke, Chris Lampton

Foreword by André LaMothe

Build Your Own Flight Sim in C++ shows you how to transform flat screens into 3D fantasy worlds. This update to the ground-breaking bestseller (previously titled *Flights of Fantasy*) covers all the latest technological breakthroughs, giving you all the tips and techniques you need to create realistic animated graphics. The CD includes source code, a sound tool kit, utilities, and a complete sample flight simulator and demos.

Available Now, 660 pages
ISBN: 1-57169-022-0
U.S. $39.99
1 – CD-ROM

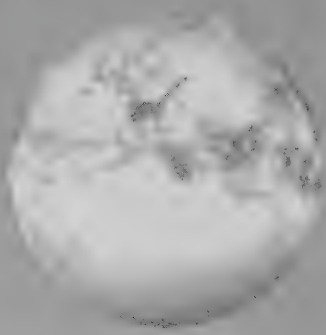


OPENGL SUPERBIBLE

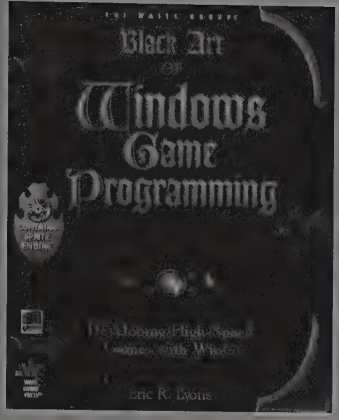
Richard S. Wright, Jr.

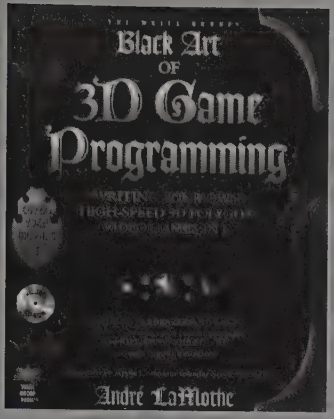
The most comprehensive, definitive guide to OpenGL programming for Windows NT and Windows 95 available. This new addition to the popular SuperBible series offers hundreds of tricks and techniques for mastering OpenGL—the powerful 3D graphics API that is the undisputed king in the special effects and entertainment industry. The CD includes all the source code and binaries for the examples in the book, as well as executable files for Intel, MIPS, ALPHA, and PowerPC-based systems, a free OCX control and OpenGL VRML Viewer.

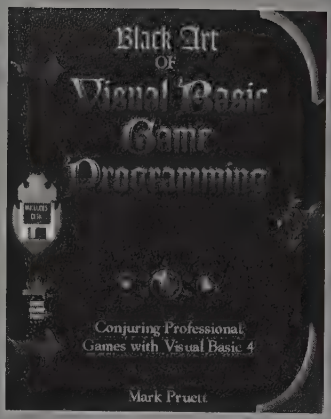
Available August, 900 pages
ISBN: 1-57169-073-5
U.S. $59.99
1 – CD-ROM



Send for our unique catalog to get more information about these books as well as our outstanding and award-winning titles.







THE BLACK ART OF WINDOWS GAME PROGRAMMING

Eric R. Lyons

The Black Art of Windows Game Programming makes it easy to use the WinG DLL to write high-performance Windows games, even if you've never developed Windows programs before. This book teaches real-world game programming techniques, featuring practical examples, hints, and code.

Available Now, 700 pages
ISBN: 1-878739-95-6
U.S. $34.95
1 – CD-ROM

THE BLACK ART OF 3D GAME PROGRAMMING

André LaMothe

This hands-on, step-by-step guide covers all essential game-writing techniques, including wire frame rendering, hidden surface removal, solid modeling and shading, input devices, artificial intelligence, modem communications, lighting, transformations, digitized VR sound and music—even voxel graphics. The CD-ROM includes a 3D modem-to-modem space combat simulator that you build from scratch.

Available Now, 1,000 pages
ISBN: 1-57169-004-2
U.S. $49.95
1 – CD-ROM

THE BLACK ART OF VISUAL BASIC GAME PROGRAMMING

Mark Pruett

The Black Art of Visual Basic Game Programming shows you how to exploit the power of Visual Basic and the Windows API to write games with blazing graphics and full sound. Step-by-step instructions and carefully explained code reveal each phase of high-end game development. A CD-ROM with the games and source code is included, so that you can add the working code modules to your own projects.

Available Now, 550 pages
ISBN: 1-57169-005-0
U.S. $34.95
1 – CD-ROM

WAITE GROUP PRESS™

TO ORDER TOLL FREE, CALL 1-800-428-5331

Source Code: MCPB

FAX 1-800-882-8583

OR SEND ORDER FORM TO: MACMILLAN COMPUTER PROGRAMMING ATTENTION: ORDER DEPARTMENT, 201 W. 103RD ST., INDIANAPOLIS, IN 46290

Qty	Book	US	Total
___	The Black Art of Java Game Programming	$49.99	_____
___	Build Your Own Flight Sim in C++	$39.99	_____
___	OpenGL SuperBible	$59.99	_____
___	The Black Art of Windows Game Programming	$34.95	_____
___	The Black Art of 3D Game Programming	$49.95	_____
___	The Black Art of VB Game Programming	$34.95	_____
Calif. residents add 7.25% Sales Tax			_____
Shipping			
USPS ($5 first book/$1 each add'l)			_____
UPS Two Day ($10/$2)			_____
Canada ($10/$4)			_____
TOTAL			_____

Ship to:

Name _____

Company _____

Address _____

City, State, Zip _____

Phone _____

Payment Method

☐ Check Enclosed ☐ Visa ☐ MasterCard

Card# _____ Exp. Date _____

Signature _____

SATISFACTION GUARANTEED OR YOUR MONEY BACK.

This is a legal agreement between you, the end user and purchaser, and The Waite Group®, Inc., and the authors of the programs contained in the disk. By opening the sealed disk package, you are agreeing to be bound by the terms of this Agreement. If you do not agree with the terms of this Agreement, promptly return the unopened disk package and the accompanying items (including the related book and other written material) to the place you obtained them for a refund.

SOFTWARE LICENSE

1. The Waite Group, Inc. grants you the right to use one copy of the enclosed software programs (the programs) on a single computer system (whether a single CPU, part of a licensed network, or a terminal connected to a single CPU). Each concurrent user of the program must have exclusive use of the related Waite Group, Inc. written materials.

2. The program, including the copyrights in each program, is owned by the respective author and the copyright in the entire work is owned by The Waite Group, Inc. and they are therefore protected under the copyright laws of the United States and other nations, under international treaties. You may make only one copy of the disk containing the programs exclusively for backup or archival purposes, or you may transfer the programs to one hard disk drive, using the original for backup or archival purposes. You may make no other copies of the programs, and you may make no copies of all or any part of the related Waite Group, Inc. written materials.

3. You may not rent or lease the programs, but you may transfer ownership of the programs and related written materials (including any and all updates and earlier versions) if you keep no copies of either, and if you make sure the transferee agrees to the terms of this license.

4. You may not decompile, reverse engineer, disassemble, copy, create a derivative work, or otherwise use the programs except as stated in this Agreement.

GOVERNING LAW

This Agreement is governed by the laws of the State of California.

LIMITED WARRANTY

The following warranties shall be effective for 90 days from the date of purchase: (i) The Waite Group, Inc. warrants the enclosed disk to be free of defects in materials and workmanship under normal use; and (ii) The Waite Group, Inc. warrants that the programs, unless modified by the purchaser, will substantially perform the functions described in the documentation provided by The Waite Group, Inc. when operated on the designated hardware and operating system. The Waite Group, Inc. does not warrant that the programs will meet purchaser's requirements or that operation of a program will be uninterrupted or error-free. The program warranty does not cover any program that has been altered or changed in any way by anyone other than The Waite Group, Inc. The Waite Group, Inc. is not responsible for problems caused by changes in the operating characteristics of computer hardware or computer operating systems that are made after the release of the programs, nor for problems in the interaction of the programs with each other or other software.

THESE WARRANTIES ARE EXCLUSIVE AND IN LIEU OF ALL OTHER WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR OF ANY OTHER WARRANTY, WHETHER EXPRESS OR IMPLIED.

EXCLUSIVE REMEDY

The Waite Group, Inc. will replace any defective disk without charge if the defective disk is returned to The Waite Group, Inc. within 90 days from date of purchase.

This is Purchaser's sole and exclusive remedy for any breach of warranty or claim for contract, tort, or damages.

LIMITATION OF LIABILITY

THE WAITE GROUP, INC. AND THE AUTHORS OF THE PROGRAMS SHALL NOT IN ANY CASE BE LIABLE FOR SPECIAL, INCIDENTAL, CONSEQUENTIAL, INDIRECT, OR OTHER SIMILAR DAMAGES ARISING FROM ANY BREACH OF THESE WARRANTIES EVEN IF THE WAITE GROUP, INC. OR ITS AGENT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

THE LIABILITY FOR DAMAGES OF THE WAITE GROUP, INC. AND THE AUTHORS OF THE PROGRAMS UNDER THIS AGREEMENT SHALL IN NO EVENT EXCEED THE PURCHASE PRICE PAID.

COMPLETE AGREEMENT

This Agreement constitutes the complete agreement between The Waite Group, Inc. and the authors of the programs, and you, the purchaser.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above exclusions or limitations may not apply to you. This limited warranty gives you specific legal rights; you may have others, which vary from state to state.

SATISFACTION REPORT CARD

Please fill out this card if you wish to know of future updates to *Spells of Fury*, or to receive our catalog.

SATISFACTION CARD

First Name: _____ **Last Name:** _____

Street Address: _____

City: _____ **State:** _____ **Zip:** _____

E-mail Address _____

Daytime Telephone: ( ) _____

Date product was acquired: **Month** **Day** **Year** **Your Occupation:**

Overall, how would you rate *Spells of Fury*?

☐ Excellent ☐ Very Good ☐ Good
☐ Fair ☐ Below Average ☐ Poor

What did you like MOST about this book? _____

What did you like LEAST about this book? _____

Please describe any problems you may have encountered with installing or using the disk: _____

How did you use this book (problem-solver, tutorial, reference...)? _____

What is your level of computer expertise?

☐ New ☐ Dabbler ☐ Hacker
☐ Power User ☐ Programmer ☐ Experienced Professional

What computer languages are you familiar with? _____

Please describe your computer hardware:

Computer _____ Hard disk _____
5.25" disk drives _____ 3.5" disk drives _____
Video card _____ Monitor _____
Printer _____ Peripherals _____
Sound Board _____ CD ROM _____

Where did you buy this book?

☐ Bookstore (name): _____
☐ Discount store (name): _____
☐ Computer store (name): _____
☐ Catalog (name): _____
☐ Direct from WGP ☐ Other _____

What price did you pay for this book? _____

What influenced your purchase of this book?

☐ Recommendation ☐ Advertisement
☐ Magazine review ☐ Store display
☐ Mailing ☐ Book's format
☐ Reputation of Waite Group Press ☐ Other

How many computer books do you buy each year? _____

How many other Waite Group books do you own? _____

What is your favorite Waite Group book? _____

Is there any program or subject you would like to see Waite Group Press cover in a similar approach? _____

Additional comments? _____

Please send to: Waite Group Press
200 Tamal Plaza
Corte Madera, CA 94925

☐ **Check here for a free Waite Group catalog**



BEFORE YOU OPEN THE DISK OR CD-ROM PACKAGE ON THE FACING PAGE, CAREFULLY READ THE LICENSE AGREEMENT.

Opening this package indicates that you agree to abide by the license agreement found in the back of this book. If you do not agree with it, promptly return the unopened disk package (including the related book) to the place you obtained them for a refund.


SPELLS OF
FURY
© 1996
Waite Group Press™
A Division of
Sams Publishing
See book for
installation instructions.

SPELLS OF FURY

Start building head-spinning, supremely powerful games with DirectX 2, the Windows 95 Game SDK. *Spells of Fury* is the insider's guide to writing high-speed, real-time games that tear up the screen with Microsoft's new DirectDraw, DirectSound, DirectInput, and DirectPlay APIs.

Step-by-step instructions clearly illustrate priceless design tips, examples, and strategies for creating professional games such as Asteroids, Bubonic, and Wolfenstein. Escalate your programming prowess using *Spells of Fury*'s full spectrum coverage of game programming essentials and advanced topics including multiple players and networking using Winsock and DirectPlay.

The companion CD-ROM contains the source code from the book, with art and .WAV sound files, ten example games that are built in the book, and a popular shareware game title.

"Definitely the most comprehensive work on the technology that I've seen to date." —*Alex St. John Microsoft*

"Brings the terrors of Windows 95 game programming down to earth." —*Jeff Bankston*

"The most complete treatise on DirectX that I have seen thus far." —*André LaMothe*

PLATFORM: PC-compatible
OPERATING SYSTEM: Windows 95
LEVEL: Intermediate-Advanced
REQUIRES: 486 or higher PC, DirectX SDK, VC++ 4, or Watcom 10.6

WAITE GROUP PRESS™
200 Tamal Plaza
Corte Madera, CA 94925

7 84376 90670 7

Spells of Fury gives you:

- **Demonstrations of the new Windows 95 APIs:** Make your games zoom and scream with DirectDraw and DirectSound
- **All the game programming essentials:** Master every key concept from bitmaps, vector graphics, and polygons to blitting, raycasting, and animation
- **Ten game examples:** Use the DirectX 2 creations on the included CD-ROM as models for constructing your own games
- **Easy-to-follow tutorials:** Absorb new Windows game programming information quickly, without any confusion
- **Visual C++ and Win32 SDK overview:** Discover how to work with palettes, design DIB classes, draw with DGI, and more
- **Detailed coverage of advanced topics:** Tackle tougher game development issues like programming multiple players and networking with Winsock and DirectPlay
- **Leading game market examples:** Sneak a peak at professional coding and design tactics behind industry hits such as Asteroids, Bubonic the WartHog, and Wolfenstein
- **Game strategy and design tips:** Uncover how the game gurus add missiles, calibrate joysticks, create 3D effects, pause game play, and more
- **Instant results:** Start game programming now with the complete source code, art, executables, plus all the book's examples on CD-ROM

$49.99 USA
$70.99 Canada

SHELVING: PROGRAMMING/GAMES
ISBN 1-57169-067-0
54999>
9 781571 690678